Borgitory Documentation

Release 1.0.0

mlapaglia

Mar 07, 2026

User Guide

1 Key Features **3**

2 Quick Start with Docker **5**

3 Quick Links **7**

4 Getting Help **9**

5 Contents **11**

5.1 Installation Guide 11
5.2 Features Overview 17
5.3 Usage Guide 26
5.4 How-To Guides 38
5.5 Troubleshooting Guide 109
5.6 Adding New Cloud Providers 121
5.7 API Reference 136
5.8 Architecture Overview 144
5.9 Changelog 161
5.10 Contributing to Borgitory 164
5.11 License 174

Borgitory is a comprehensive web-based management interface for BorgBackup repositories with real-time monitoring, automated scheduling, and cloud synchronization capabilities.

Built with modern web technologies, Borgitory provides an intuitive interface for managing your backup infrastructure, whether you're running a single repository or managing multiple backup destinations across different storage providers.

1 Key Features

- **Repository Management**: Add, configure, and manage multiple Borg repositories
- **Real-time Monitoring**: Live backup progress with detailed task tracking
- **Automated Scheduling**: Cron-based backup scheduling with integrated cleanup
- **Archive Browser**: Interactive exploration of backup archives with FUSE mounting
- **Cloud Synchronization**: Multi-provider cloud sync using Rclone
- **Push Notifications**: Pushover integration for job completion alerts
- **Modern UI**: Responsive interface built with HTMX, Alpine.js, and Tailwind CSS

2 Quick Start with Docker

The easiest way to get started with Borgitory is using Docker:

```
# Pull the latest image
docker pull mlapaglia/borgitory:latest

# Run Borgitory
docker run -d \
  --name borgitory \
  -p 8000:8000 \
  -v /path/to/your/repos:/repos \
  -v borgitory-data:/app/data \
  mlapaglia/borgitory:latest

# Access at http://localhost:8000
```

For production deployments and advanced configuration, see the *Installation Guide* guide.

Note

Borgitory requires BorgBackup to be available. The Docker image includes BorgBackup and Rclone pre-installed.

3 Quick Links

- *Installation Guide* - Docker deployment and alternative installation methods
- *Features Overview* - Comprehensive feature overview
- *Usage Guide* - Step-by-step usage guide
- *How-To Guides* - Task-specific guides and recipes
- *API Reference* - REST API documentation

4 Getting Help

- GitHub Issues - Report bugs or request features
- GitHub Discussions - Community support and questions
- *Troubleshooting Guide* - Common issues and solutions

5

Contents

5.1 Installation Guide

Borgitory can be installed in multiple ways depending on your preferences and environment. **Docker is the recommended installation method** as it provides the easiest setup with all dependencies included. This guide covers all installation methods and their requirements.

5.1.1 Prerequisites

Before installing Borgitory, ensure you have the following prerequisites:

System Requirements

- **Operating System**: Linux, macOS, or Windows
- **Python**: 3.11 or higher (for PyPI installation)
- **Docker**: Docker with Docker Compose (for containerized deployment)

> **Note**
>
> **Windows Users:** For full functionality including archive browsing, Docker installation is strongly recommended as FUSE (required for archive browsing) is not available on Windows.
>
> Alternatively, you can use WSL2 (Windows Subsystem for Linux) to run Borgitory natively. See the *Windows WSL Setup Guide* guide for complete WSL2 setup instructions.

5.1.2 Installation Methods

Method 1: Docker Installation (Recommended)

Docker is the recommended way to run Borgitory as it provides an isolated environment with all dependencies pre-configured:

Using Docker Directly

```
# Pull and run the Docker image
docker run -d \
  -p 8000:8000 \
  -v ./data:/app/data \
  -v /path/to/backup/sources:/mnt/backup/sources:ro \
  -v /path/to/borg/repos:/mnt/repos \
  --cap-add SYS_ADMIN \
  --device /dev/fuse \
  --name borgitory \
  mlapaglia/borgitory:latest
```

Using Docker Compose (Recommended)

Create a `docker-compose.yml` file:

```
version: '3.8'
services:
  borgitory:
    image: mlapaglia/borgitory:latest
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data
      - /path/to/backup/sources:/mnt/backup/sources:ro
      - /path/to/borg/repos:/mnt/repos
    cap_add:
      - SYS_ADMIN
    devices:
      - /dev/fuse
    restart: unless-stopped
```

Then start the container:

```
docker-compose up -d
```

Docker Installation Benefits:

- All dependencies (BorgBackup, Rclone, FUSE) pre-installed
- Isolated environment with consistent behavior
- Easy updates and rollbacks
- No system-level dependency conflicts

Method 2: PyPI Installation

For users who prefer a native Python installation:

```
# Install stable release from PyPI
pip install borgitory

# Start the server
```

(continues on next page)

(continued from previous page)

```
borgitory serve

# Or run with custom settings
borgitory serve --host 0.0.0.0 --port 8000
```

PyPI Installation Requirements:

- Python 3.11 or higher
- BorgBackup installed and available in PATH
- Rclone (optional, for cloud sync features)
- FUSE (Linux/macOS only, for archive browsing)

> **Note**
>
> PyPI installation requires manual setup of BorgBackup and Rclone dependencies. For easier setup, consider using the Docker installation method.

> **Warning**
>
> **Windows Limitation:** FUSE (Filesystem in Userspace) is not available on Windows, which means archive browsing functionality will not work with PyPI installation on Windows. The Docker installation includes a Linux environment where FUSE works properly, making it the recommended approach for Windows users who need archive browsing.

Method 3: Development Installation

For development or contributing to Borgitory:

```
# Clone the repository
git clone https://github.com/mlapaglia/Borgitory.git
cd Borgitory

# Create virtual environment
python -m venv .env_borg

# Activate virtual environment
# On Windows:
.env_borg\Scripts\activate
# On macOS/Linux:
source .env_borg/bin/activate

# Install in development mode
pip install -e .[dev]

# Run development server
python run.py
```

5.1.3 Docker Configuration

Volume Mapping Strategy

All volumes must be mounted under `/mnt/` to be visible in the application:

```
volumes:
  - ./data:/app/data                                  # Persistent application data␣
↪(required)
  - /path/to/backup/sources:/mnt/backup/sources:ro # Source directories (read-only)
  - /path/to/borg/repos:/mnt/repos                    # Repository storage (read-write)
  - /additional/source:/mnt/additional:ro             # Additional sources as needed
  - /another/repo/location:/mnt/alt-repos             # Additional repositories as needed
```

Volume Guidelines:

- Mount as many volumes as necessary for all backup sources and repository locations
- Source directories can be mounted read-only (`:ro`) for safety
- Repository directories need read-write access for Borg operations
- Each volume can be mapped to any convenient path under `/mnt/` inside the container
- Supports distributed setups where repositories and sources are in different locations

Required Docker Parameters

For full functionality, the following Docker parameters are required:

`--cap-add SYS_ADMIN`
: Required for FUSE filesystem mounting to enable the archive browser feature

`--device /dev/fuse`
: Provides access to the FUSE device for archive filesystem mounting

FUSE Requirements:

- Enables the interactive archive browser feature
- Allows real-time exploration of backup archives without extraction
- Supports direct file downloads from mounted archive filesystems
- Without FUSE support, archive browsing will be disabled

5.1.4 Installing Dependencies

BorgBackup Installation

Ubuntu/Debian:

```
sudo apt update
sudo apt install borgbackup
```

CentOS/RHEL/Fedora:

```
sudo dnf install borgbackup
```

macOS:

```
brew install borgbackup
```

Windows:

Download from the BorgBackup releases page

> **Note**
>
> For Windows users who want full functionality including FUSE-based archive browsing, we recommend using WSL2. See *Windows WSL Setup Guide* for a complete setup guide.

Rclone Installation (Optional)

For cloud synchronization features:

Linux:

```
curl https://rclone.org/install.sh | sudo bash
```

macOS:

```
brew install rclone
```

Windows:

Download from the Rclone downloads page

5.1.5 First-Time Setup

After installation, access the web interface:

1. Open http://localhost:8000 in your browser
2. Create your first admin account on the initial setup page
3. Configure your first repository in the dashboard

The application will create a local SQLite database and generate encryption keys automatically.

5.1.6 Configuration Options

Environment Variables

Borgitory can be configured using environment variables:

`BORGITORY_HOST`
: Server host address (default: `0.0.0.0`)

`BORGITORY_PORT`
: Server port (default: `8000`)

`BORGITORY_DATA_DIR`
: Data directory path (default: `./data`)

`BORGITORY_DEBUG`
: Enable debug mode (default: `false`)

Command Line Options

When using the PyPI installation:

```
borgitory serve --help

Options:
  --host TEXT      Host to bind to [default: 0.0.0.0]
  --port INTEGER   Port to bind to [default: 8000]
  --reload         Enable auto-reload for development
  --log-level TEXT Log level [default: info]
  --help           Show this message and exit
```

5.1.7 Verification

To verify your installation is working correctly:

1. **Check the web interface**: Navigate to http://localhost:8000
2. **Verify BorgBackup**: The dashboard will show if BorgBackup is available
3. **Test repository creation**: Try adding a test repository
4. **Check logs**: Monitor the application logs for any errors

Docker verification:

```
# Check container status
docker-compose ps

# View logs
docker-compose logs -f borgitory
```

PyPI verification:

```
# Check if borgitory command is available
borgitory --help

# Verify BorgBackup is available
borg --version
```

5.1.8 Troubleshooting Installation

Common Issues

Python Version Issues
Ensure you're using Python 3.11 or higher:

```
python --version
```

BorgBackup Not Found
Verify BorgBackup is installed and in PATH:

```
borg --version
which borg
```

Docker Permission Issues
Ensure your user has permission to run Docker commands:

```
sudo usermod -aG docker $USER
# Log out and back in
```

FUSE Mount Issues

On some systems, you may need to install FUSE utilities:

```
# Ubuntu/Debian
sudo apt install fuse3

# CentOS/RHEL/Fedora
sudo dnf install fuse3
```

For more troubleshooting information, see the *Troubleshooting Guide* guide.

5.1.9 Next Steps

After successful installation:

1. Read the *Features Overview* overview to understand Borgitory's capabilities
2. Follow the *Usage Guide* guide for step-by-step instructions
3. Explore *How-To Guides* guides for specific tasks
4. Configure *Adding New Cloud Providers* for cloud synchronization

5.2 Features Overview

Borgitory provides a comprehensive set of features for managing BorgBackup repositories through a modern web interface. This page details all available functionality and capabilities.

5.2.1 Core Backup Management

Repository Management

Multi-Repository Support

- Add and manage multiple Borg repositories simultaneously
- Support for local and remote repositories
- Secure passphrase storage with encryption
- Repository validation and connection testing
- Repository statistics and health monitoring

Repository Configuration

- Flexible repository paths (local filesystem or network locations)
- Encrypted credential storage using Fernet encryption
- Repository-specific settings and preferences
- Automatic repository discovery and validation

Manual Backup Operations

On-Demand Backups

- Create backups instantly with configurable parameters
- Flexible source path selection with path validation
- Multiple compression algorithms (lz4, zlib, lzma, zstd)
- Custom archive naming with timestamp support
- Advanced include/exclude patterns with multiple styles (see *Backup Patterns Guide*)

Real-Time Progress Monitoring

- Live backup progress with detailed statistics
- File-by-file progress tracking
- Transfer rate and ETA calculations
- Expandable task details with full command output
- Server-Sent Events for real-time updates

Archive Management

Archive Browser

- Interactive directory-based archive exploration
- FUSE-mounted filesystem for direct file access
- Real-time navigation through archive contents
- File metadata display (size, modification dates, permissions)
- Breadcrumb navigation for easy directory traversal

File Downloads

- Direct file downloads from mounted archives
- Streaming downloads without temporary storage
- Support for large files and slow connections
- Multiple concurrent downloads
- Efficient FUSE-based file access

Archive Operations

- Archive listing and metadata viewing
- Archive deletion and cleanup
- Archive information and statistics
- Archive comparison and diff capabilities

5.2.2 Automated Operations

Scheduled Backups

Cron-Based Scheduling

- Flexible cron expression support for backup timing

- Human-readable cron descriptions
- Multiple schedules per repository
- Schedule enable/disable functionality
- Timezone-aware scheduling

Advanced Scheduling Options

- Custom compression settings per schedule
- Source path configuration per schedule
- Integration with pruning policies
- Notification settings per schedule
- Schedule conflict detection and management

Archive Pruning

Retention Policies

- **Simple Strategy**: Keep archives within X days
- **Advanced Strategy**: Granular retention rules
  - Daily archives retention
  - Weekly archives retention
  - Monthly archives retention
  - Yearly archives retention

Pruning Options

- Dry-run mode to preview pruning actions
- Detailed prune lists showing what will be removed
- Space savings calculations and statistics
- Force prune execution for immediate cleanup
- Integration with backup schedules

Pruning Safety Features

- Preview mode before actual deletion
- Comprehensive logging of pruning operations
- Space reclamation reporting
- Rollback capabilities for safety

5.2.3 Cloud Synchronization

Multi-Provider Support

Borgitory supports multiple cloud storage providers through Rclone integration:

Currently Supported Providers:

- Amazon S3 and S3-compatible services
- Google Cloud Storage

- Microsoft Azure Blob Storage
- Backblaze B2
- DigitalOcean Spaces
- Wasabi Hot Cloud Storage
- Generic S3-compatible services

Provider Features:

- Automatic provider discovery through registry system
- Provider-specific configuration validation
- Connection testing and validation
- Secure credential storage
- Provider-specific optimization settings

Cloud Sync Operations

Automated Synchronization

- Post-backup automatic sync to cloud storage
- Manual sync triggers for immediate uploads
- Progress tracking with real-time updates
- Retry logic for failed transfers
- Bandwidth limiting and throttling options

Sync Configuration

- Path prefix configuration for organized storage
- Encryption settings for cloud storage
- Compression options for cloud transfers
- Selective sync patterns and filtering
- Multi-destination sync support

Sync Monitoring

- Real-time sync progress with transfer statistics
- Sync history and logging
- Error reporting and retry mechanisms
- Bandwidth usage monitoring
- Success/failure notifications

5.2.4 Job Management System

Real-Time Job Monitoring

Live Job Tracking

- Real-time job status updates
- Progress indicators for each job stage

- Expandable task details with full output
- Job queuing and prioritization
- Concurrent job management

Job History

- Persistent storage of all job results
- Searchable job history with filtering
- Detailed job logs and error reporting
- Job duration and performance metrics
- Export capabilities for job data

Task Management

- Individual task tracking within jobs
- Task dependency management
- Task retry and recovery mechanisms
- Task output streaming and logging
- Task cancellation and cleanup

5.2.5 Notifications

Push Notifications

Pushover Integration

- Job completion notifications
- Failure alerts with error details
- Success confirmations
- Configurable notification triggers
- Custom message formatting

Notification Settings

- Per-schedule notification configuration
- Global notification preferences
- Notification filtering and rules
- Emergency notification escalation
- Quiet hours and notification scheduling

5.2.6 User Interface

Modern Web Interface

Technology Stack

- **HTMX**: Dynamic HTML updates without JavaScript frameworks
- **Alpine.js**: Lightweight JavaScript reactivity for interactive components
- **Tailwind CSS**: Utility-first styling with responsive design

- **Server-Sent Events**: Real-time progress updates and live monitoring

Responsive Design

- Mobile-friendly interface with touch optimization
- Tablet and desktop layout optimization
- Dark/light theme support
- Accessibility features and keyboard navigation
- Progressive web app capabilities

User Experience Features

- Intuitive navigation with breadcrumbs
- Context-sensitive help and tooltips
- Drag-and-drop file operations
- Keyboard shortcuts for power users
- Customizable dashboard layouts

5.2.7 Security Features

Authentication & Authorization

User Management

- Secure username/password authentication
- BCrypt password hashing with salt
- Session management with secure cookies
- Account lockout protection
- Password strength requirements

Data Security

- Encrypted credential storage using Fernet encryption
- Secure session management
- CSRF protection on all forms
- Input validation and sanitization
- SQL injection prevention

Access Control

- Role-based access control (planned)
- Repository-level permissions (planned)
- Audit logging for security events
- Secure API endpoints with authentication
- Rate limiting and abuse prevention

5.2.8 API & Integration

REST API

Comprehensive API

- Full REST API for all functionality
- OpenAPI/Swagger documentation at `/docs`
- ReDoc documentation at `/redoc`
- JSON-based request/response format
- Authentication via API keys (planned)

API Features

- Repository management endpoints
- Backup operation triggers
- Job monitoring and history
- Configuration management
- Real-time status endpoints

Integration Support

- Webhook support for external integrations
- CLI tool for automation scripts
- Docker API for container management
- Prometheus metrics export (planned)
- Third-party monitoring system integration

5.2.9 Monitoring & Observability

System Monitoring

Health Checks

- Application health monitoring
- Repository connectivity checks
- Storage space monitoring
- Service dependency validation
- Performance metrics collection

Logging & Debugging

- Comprehensive application logging
- Structured logging with JSON format
- Log level configuration
- Debug mode for troubleshooting
- Log rotation and archival

Performance Monitoring

- Job execution time tracking
- Resource usage monitoring
- Transfer rate optimization
- Bottleneck identification
- Performance trend analysis

5.2.10 Architecture Features

Modern Python Stack

Backend Technologies

- **FastAPI**: Modern Python web framework with automatic OpenAPI docs
- **SQLAlchemy**: Powerful ORM with async support
- **APScheduler**: Advanced job scheduling with cron support
- **Pydantic**: Data validation and serialization
- **Alembic**: Database migrations and versioning

Database & Storage

- SQLite for lightweight deployment
- PostgreSQL support (planned)
- File-based configuration storage
- Encrypted sensitive data storage
- Automatic database migrations

Scalability Features

- Async/await throughout the application
- Connection pooling and optimization
- Background task processing
- Horizontal scaling support (planned)
- Container orchestration ready

5.2.11 Development & Deployment

Deployment Options

Container Support

- Official Docker images on Docker Hub
- Docker Compose configurations
- Kubernetes deployment manifests (planned)
- Multi-architecture container support
- Optimized container layers

Installation Methods

- PyPI package installation

- Docker containerized deployment
- Development installation from source
- Automated deployment scripts
- Configuration management integration

Development Features

- Hot-reload development server
- Comprehensive test suite with pytest
- Code linting and formatting with Ruff
- Type checking with MyPy
- GitHub Actions CI/CD pipeline

Extensibility

Plugin System (Planned)

- Custom provider plugins
- Notification plugin architecture
- Custom authentication providers
- Storage backend extensions
- UI theme and customization support

Configuration Management

- Environment variable configuration
- Configuration file support
- Runtime configuration updates
- Configuration validation
- Template-based configuration generation

5.2.12 Coming Soon

Planned Features

- Multi-user support with role-based access control
- PostgreSQL database backend support
- Advanced monitoring with Prometheus metrics
- Plugin system for extensibility
- Kubernetes operator for orchestration
- Advanced reporting and analytics
- Backup verification and integrity checking
- Multi-tenancy support
- Advanced search and filtering
- Backup deduplication analysis

For the latest feature updates and roadmap, visit the GitHub repository.

5.3 Usage Guide

This guide walks you through using Borgitory's key features step-by-step, from initial setup to advanced operations. Examples assume you're using the recommended Docker installation method.

5.3.1 Getting Started

First-Time Setup

After starting Borgitory with Docker, access the web interface at http://localhost:8000:

```
# Start Borgitory (if not already running)
docker run -d \
  --name borgitory \
  -p 8000:8000 \
  -v borgitory-data:/app/data \
  mlapaglia/borgitory:latest
```

Now you can access the web interface:

1. **Create Admin Account**
   - On first visit, you'll see the initial setup page
   - Create your admin username and password
   - Click "Create Account" to complete setup
2. **Dashboard Overview**
   - The main dashboard shows repository status
   - Navigation menu provides access to all features
   - Quick actions are available in the toolbar

5.3.2 Repository Management

Adding Your First Repository

1. **Navigate to Repositories**
   - Click "Repositories" in the main navigation
   - Click "Add Repository" button
2. **Configure Repository**

   ```
   Name: My Backup Repo
   Path: /mnt/repos/my-backup-repo
   Passphrase: [secure-passphrase]
   ```

   - **Name**: Friendly identifier for the repository
   - **Path**: Full path to repository location (must be under /mnt/ for Docker)
   - **Passphrase**: Encryption password for the repository
3. **Test Connection**

- Click "Test Connection" to verify repository access
- Green checkmark indicates successful connection
- Red X indicates configuration issues

4. **Save Repository**
   - Click "Save" to add the repository
   - Repository appears in the main dashboard

Managing Multiple Repositories

Repository List View

- View all configured repositories
- See connection status and last backup time
- Quick actions: Edit, Delete, Test Connection

Repository Details

- Click repository name to view detailed information
- Repository statistics and health metrics
- Recent backup history and archive list

5.3.3 Creating Backups

Manual Backup

1. **Start New Backup**
   - From dashboard, click "New Backup" button
   - Or navigate to Backups → Manual Backup
2. **Configure Backup Settings**

```
Repository: My Backup Repo
Source Path: /mnt/backup/sources/documents
Archive Name: documents-{now:%Y%m%d-%H%M%S}
Compression: lz4
```

   - **Repository**: Select target repository from dropdown
   - **Source Path**: Directory to backup (must be under /mnt/ for Docker)
   - **Archive Name**: Name template with timestamp support
   - **Compression**: Choose compression algorithm (lz4, zlib, lzma, zstd)
3. **Advanced Options** (Optional)
   - **Include/Exclude Patterns**: File patterns to include/exclude from backup (see *Backup Patterns Guide* for detailed guide)
   - **One File System**: Don't cross filesystem boundaries
   - **Numeric Owner**: Store numeric user/group IDs
   - **Checkpoint Interval**: Create checkpoints every N seconds
4. **Start Backup**

- Click "Start Backup" to begin the process
- Real-time progress appears immediately
- Monitor transfer rates and file counts

Monitoring Backup Progress

Real-Time Progress Display

- Live progress bar with percentage complete
- Current file being processed
- Transfer rate and estimated time remaining
- Total files processed and data transferred

Expandable Task Details

- Click "Show Details" to view full command output
- See Borg's detailed progress information
- Monitor any warnings or errors
- View compression statistics

Job Completion

- Success notification with backup summary
- Archive information and statistics
- Links to browse the new archive
- Option to start cloud sync if configured

5.3.4 Scheduled Backups

Creating Backup Schedules

1. **Navigate to Schedules**
   - Click "Schedules" in the main navigation
   - Click "Add Schedule" button
2. **Basic Schedule Configuration**

```
Name: Daily Documents Backup
Repository: My Backup Repo
Source Path: /mnt/backup/sources/documents
Cron Expression: 0 2 * * *
Enabled: ✓
```

   - **Name**: Descriptive name for the schedule
   - **Repository**: Target repository for backups
   - **Source Path**: Directory to backup regularly
   - **Cron Expression**: When to run (daily at 2 AM in this example)
   - **Enabled**: Toggle to activate/deactivate schedule
3. **Advanced Schedule Options**

- **Compression**: Set compression algorithm
- **Archive Naming**: Template for archive names
- **Pruning Policy**: Attach retention policy
- **Notifications**: Configure success/failure alerts
- **Cloud Sync**: Enable automatic cloud synchronization
- **Job Hooks**: Configure pre-job and post-job automation scripts (see *Package Manager for Pre/Post Hooks* for installing required tools)

Understanding Cron Expressions

Common cron patterns:

```
0 2 * * *     # Daily at 2:00 AM
0 2 * * 0     # Weekly on Sunday at 2:00 AM
0 2 1 * *     # Monthly on 1st at 2:00 AM
0 */6 * * *   # Every 6 hours
30 1 * * 1-5  # Weekdays at 1:30 AM
```

The interface shows human-readable descriptions of cron expressions.

Job Hooks System

Job hooks allow you to execute custom commands before and after backup jobs, enabling powerful automation workflows.

Hook Types

- **Pre-Job Hooks**: Execute before the backup process begins
  - Ideal for database dumps, stopping services, mounting drives
  - Can be marked as "Critical" to stop the job if they fail
  - Prepare systems for consistent backups
- **Post-Job Hooks**: Execute after the backup process completes
  - Perfect for cleanup, service restarts, health checks
  - Can be configured to run even if the job fails
  - Restore system state and send notifications

Adding Hooks to a Schedule

1. **Open Hooks Configuration**
   - In the schedule creation/editing form, click "Configure Hooks"
   - This opens the hooks configuration modal
2. **Configure Pre-Job Hooks**

```
Name: Database Dump
Command: pg_dump myapp > /tmp/myapp_backup.sql
Critical: ✓ Enabled

Name: Stop Web Service
```

(continues on next page)

(continued from previous page)

```
Command: systemctl stop nginx && sleep 5
Critical: ✓ Enabled
```

3. **Configure Post-Job Hooks**

```
Name: Restart Services
Command: systemctl start nginx && systemctl start mysql
Run Even If Job Failed: ✓ Enabled

Name: Health Check Ping
Command: curl -X POST https://healthcheck.io/ping/abc123
Run Even If Job Failed: ✓ Enabled
```

Hook Options

- **Critical Hooks**: Job fails immediately if a critical hook fails
  - Use for essential preparation tasks (database dumps, prerequisites)
  - Prevents inconsistent backups when preparation fails
- **Run on Job Failure** (Post-hooks only): Hook executes even if backup fails
  - Use for cleanup, service restarts, monitoring pings
  - Ensures system recovery regardless of backup success

Hook Execution Flow

Normal execution (all successful):

```
1. Pre-Hook 1 (Database Dump)        →  Success
2. Pre-Hook 2 (Stop Services)        →  Success
3. Backup Task                        →  Success
4. Prune Task                         →  Success
5. Post-Hook 1 (Restart Services)    →  Success
6. Post-Hook 2 (Health Check)        →  Success

Result:  Job Completed Successfully
```

Critical hook failure:

```
1. Pre-Hook 1 (Database Dump)        →  Failed (Critical)
2. Pre-Hook 2 (Stop Services)        →   Skipped
3. Backup Task                        →   Skipped
4. Prune Task                         →   Skipped
5. Post-Hook 1 (Restart Services)    →   Skipped
6. Post-Hook 2 (Health Check)        →  Success (Run on Failure)

Result:  Job Failed - Critical Hook Error
```

Environment Variables

Hook scripts automatically receive job context:

```
# Available in all hook scripts
BORGITORY_REPOSITORY_ID="123"     # Repository being backed up
```

(continues on next page)

(continued from previous page)

```
BORGITORY_TASK_INDEX="2"          # Hook position in job sequence
BORGITORY_JOB_TYPE="scheduled"    # Job type (scheduled, manual)
```

Common Hook Examples

Database Preparation:

```
# Pre-hook: Create PostgreSQL dump
# Requires: postgresql-client package (install via Package Manager)
#!/bin/bash
pg_dump -h localhost -U backup_user myapp > /tmp/db_backup.sql
if [ $? -ne 0 ]; then
    echo "Database dump failed"
    exit 1
fi
```

Service Management:

```
# Post-hook: Restart services (always run)
#!/bin/bash
systemctl start nginx
systemctl start mysql
echo "Services restarted"
```

Health Monitoring:

```
# Post-hook: Send health check ping
# Requires: curl package (install via Package Manager)
#!/bin/bash
curl -X POST "https://healthcheck.io/ping/your-uuid" \
     -d "Backup completed for repo ${BORGITORY_REPOSITORY_ID}"
```

For comprehensive hook documentation, examples, and troubleshooting, see *Job Hooks System Documentation*.

Managing Schedules

Schedule List

- View all configured schedules
- See next run time and last execution
- Quick enable/disable toggles
- Edit and delete options

Schedule History

- Click schedule name to view execution history
- See successful and failed runs
- Access logs and error details
- Performance metrics and trends

5.3.5 Archive Pruning

Creating Pruning Policies

1. **Navigate to Cleanup**
   - Click "Cleanup" in the main navigation
   - Click "Add Pruning Policy" button
2. **Simple Retention Strategy**

```
Name: Keep 30 Days
Strategy: Simple
Keep Days: 30
Show Details: ✓
Show Stats: ✓
```

   - **Keep Days**: Number of days to retain archives
   - **Show Details**: Display detailed list of archives to prune
   - **Show Stats**: Show space savings calculations
3. **Advanced Retention Strategy**

```
Name: Granular Retention
Strategy: Advanced
Keep Daily: 7
Keep Weekly: 4
Keep Monthly: 6
Keep Yearly: 2
```

   - **Keep Daily**: Recent daily archives to retain
   - **Keep Weekly**: Weekly archives to retain
   - **Keep Monthly**: Monthly archives to retain
   - **Keep Yearly**: Yearly archives to retain

Executing Pruning Operations

Manual Pruning

1. Select repository from dropdown
2. Choose pruning policy
3. Click "Preview Prune" to see what will be deleted
4. Review the prune list and space savings
5. Click "Execute Prune" to perform cleanup

Automated Pruning

- Attach pruning policies to backup schedules
- Pruning runs automatically after successful backups
- Monitor pruning results in job history

5.3.6 Archive Browsing

Exploring Archive Contents

1. **Access Archive Browser**
   - From dashboard, click "View Contents" next to any archive
   - Or navigate to Archives → Browse Archives
2. **Navigate Directory Structure**
   - Click folder names to navigate into directories
   - Use breadcrumb navigation to go back
   - See file sizes, modification dates, and permissions
3. **File Operations**
   - **Download**: Click download button () next to files
   - **View Details**: See file metadata and properties
   - **Search**: Use search box to find specific files

FUSE Requirements

Archive browsing requires FUSE support:

- Docker: Run with `--cap-add SYS_ADMIN --device /dev/fuse`
- Native: Ensure FUSE is installed and accessible
- Without FUSE: Archive browsing will be disabled

Downloading Files from Archives

Direct Downloads

- Files stream directly from mounted archives
- No temporary storage required
- Works efficiently with large files
- Multiple downloads can run simultaneously

Download Process

1. Navigate to desired file in archive browser
2. Click download button next to file
3. Browser starts download immediately
4. Monitor download progress in browser

5.3.7 Cloud Synchronization

Configuring Cloud Providers

1. **Navigate to Cloud Sync**
   - Click "Cloud Sync" in the main navigation
   - Click "Add Cloud Configuration" button
2. **Select Provider**

- Choose from supported providers (S3, Google Cloud, Azure, etc.)
- Provider-specific fields appear automatically

3. **Configure S3 Example**

```
Name: My S3 Backup
Provider: s3
Access Key ID: AKIAIOSFODNN7EXAMPLE
Secret Access Key: [hidden]
Bucket Name: my-backup-bucket
Region: us-east-1
Path Prefix: borgitory-backups/
```

- **Access Keys**: AWS credentials for bucket access
- **Bucket Name**: S3 bucket for storing backups
- **Region**: AWS region for the bucket
- **Path Prefix**: Organization path within bucket

4. **Test Connection**
   - Click "Test Connection" to verify configuration
   - Green checkmark indicates successful connection
   - Fix any configuration issues before saving

Manual Cloud Sync

Sync Repository to Cloud

1. Navigate to Cloud Sync → Manual Sync
2. Select repository to sync
3. Choose cloud configuration
4. Click "Start Sync" to begin upload
5. Monitor real-time sync progress

Sync Progress Monitoring

- Real-time transfer statistics
- Files uploaded and transfer rates
- Estimated time remaining
- Error reporting and retry logic

Automated Cloud Sync

Schedule Integration

- Enable cloud sync in backup schedules
- Automatic sync after successful backups
- Configure sync settings per schedule
- Monitor sync results in job history

Cloud Sync History

- View all sync operations
- See successful and failed syncs
- Access detailed sync logs
- Monitor bandwidth usage over time

5.3.8 Push Notifications

Configuring Pushover Notifications

1. **Get Pushover Credentials**
   - Sign up at https://pushover.net/
   - Create application to get API token
   - Note your user key from account settings
2. **Configure in Borgitory**
   - Navigate to Notifications
   - Click "Add Notification Configuration"
   - Select "Pushover" as provider

```
Name: My Phone Alerts
Provider: Pushover
User Key: [your-user-key]
API Token: [your-app-token]
Device: [optional-device-name]
```

3. **Test Notifications**
   - Click "Test Notification" to send test message
   - Verify notification appears on your device
   - Adjust settings if needed

Notification Settings

Global Settings

- Default notification preferences
- Quiet hours configuration
- Emergency escalation rules
- Message formatting options

Per-Schedule Settings

- Attach notifications to specific schedules
- Configure success/failure triggers
- Custom message templates
- Priority levels and sounds

5.3.9 Job Management

Monitoring Active Jobs

Jobs Dashboard

- View all active and recent jobs
- Real-time status updates
- Progress indicators for running jobs
- Quick actions: Cancel, View Details, Retry

Job Details View

- Expandable task details with full output
- Real-time log streaming
- Performance metrics and statistics
- Error reporting and diagnostics

Job History

Historical Job Data

- Search and filter job history
- View job duration and performance trends
- Access detailed logs and error reports
- Export job data for analysis

Job Analysis

- Success/failure rates over time
- Performance trending and optimization
- Resource usage patterns
- Bottleneck identification

5.3.10 Troubleshooting Common Issues

Quick Diagnostics

Repository Issues

- Verify repository path is correct and accessible
- Check passphrase is correct
- Ensure Borg is installed and in PATH
- Test repository with Borg CLI directly

Backup Failures

- Check source path exists and is readable
- Verify sufficient disk space
- Review exclude patterns for conflicts
- Check file permissions and access rights

Cloud Sync Problems

- Test cloud provider credentials
- Verify bucket/container exists
- Check network connectivity
- Review Rclone configuration

For detailed troubleshooting information, see the *Troubleshooting Guide* guide.

5.3.11 Best Practices

Repository Management

- Use descriptive repository names
- Store repositories on reliable storage
- Regular repository integrity checks
- Keep passphrases secure and backed up
- Monitor repository size growth

Backup Strategy

- Test backup and restore procedures regularly
- Use appropriate compression for your data
- Implement 3-2-1 backup strategy (3 copies, 2 different media, 1 offsite)
- Monitor backup success rates and performance
- Document your backup procedures

Security Considerations

- Use strong passphrases for repositories
- Secure cloud provider credentials
- Regular security updates and patches
- Monitor access logs for suspicious activity
- Implement proper network security

5.3.12 Next Steps

- Explore *How-To Guides* guides for specific tasks
- Configure *Adding New Cloud Providers* for additional storage options
- Review *Troubleshooting Guide* for common issues
- Check the *API Reference* documentation for automation
- Join the community on GitHub for support and updates

5.4 How-To Guides

These guides provide step-by-step instructions for specific tasks and common scenarios in Borgitory.

5.4.1 How to Set Up Multiple Repositories

This guide explains how to configure and manage multiple Borg repositories in Borgitory for different backup strategies and data organization.

Coming Soon

This guide will cover:

- Repository organization strategies
- Separating different data types
- Managing multiple repository configurations
- Cross-repository operations
- Performance considerations

For now, please refer to the *Usage Guide* guide for basic repository management.

5.4.2 How to Migrate Repositories

This guide covers migrating existing Borg repositories to Borgitory and moving repositories between storage locations.

Coming Soon

This guide will cover:

- Importing existing Borg repositories
- Moving repositories to new locations
- Updating repository configurations
- Data migration strategies
- Verification procedures

For now, please refer to the *Usage Guide* guide for repository management basics.

5.4.3 Repository Maintenance Guide

This guide covers routine maintenance tasks to keep your Borg repositories healthy and performing well.

Coming Soon

This guide will cover:

- Repository integrity checks
- Compacting repositories
- Repairing corrupted repositories
- Performance optimization
- Storage cleanup procedures

For now, please refer to the *Troubleshooting Guide* guide for common repository issues.

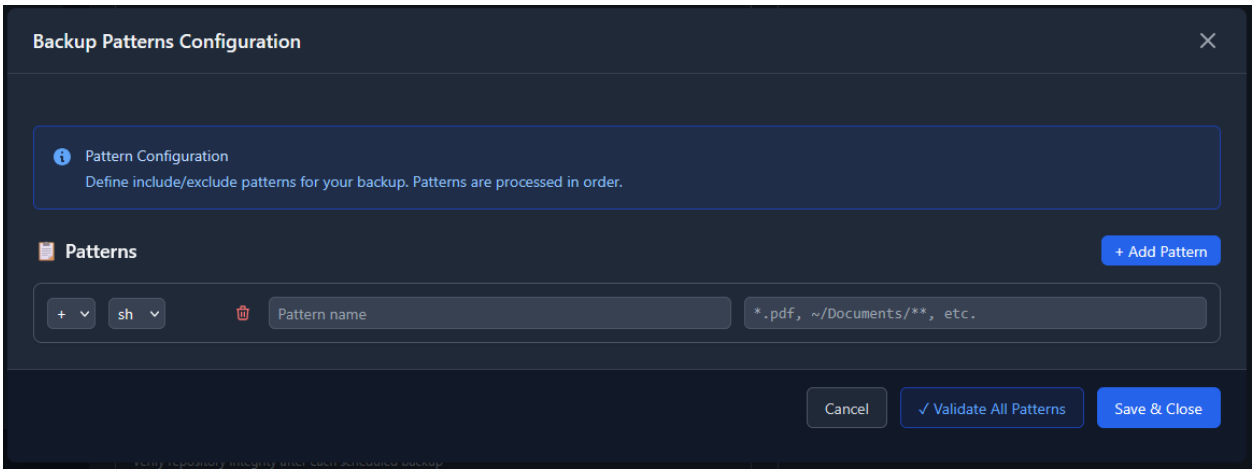
5.4.4 Backup Patterns Guide

Borgitory uses BorgBackup's powerful pattern system to control which files and directories are included or excluded from your backups. This guide explains how to configure and use backup patterns effectively.

> **Note**
>
> This guide covers Borgitory's pattern interface. For complete technical details about BorgBackup patterns, refer to the official BorgBackup pattern documentation.

Pattern Configuration Interface

Borgitory provides an intuitive web interface for configuring backup patterns. Access the pattern configuration through the backup job setup or repository configuration.

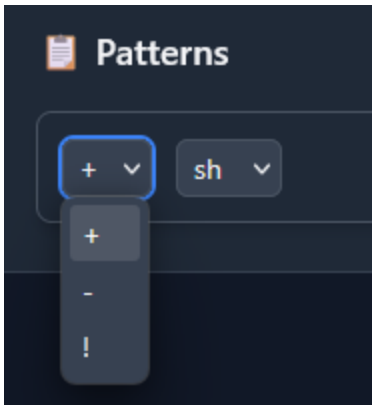


The pattern configuration interface includes:

- **Pattern list**: Shows all configured patterns in processing order
- **Add Pattern button**: Creates new include/exclude patterns
- **Action selector**: Choose include (+), exclude (-), or exclude with no recursion (!)
- **Style selector**: Select pattern matching style (sh, fm, re, pp, pf)
- **Pattern field**: Enter the actual pattern to match
- **Validation**: Real-time pattern validation and preview

Pattern Actions

Each pattern must specify an action that determines how matching files are handled:



Include (+)

Action: Include matching files and directories in the backup.

- Files matching this pattern will be backed up
- Useful for explicitly including specific files or directories
- Takes precedence over exclude patterns when there are conflicts

Example: `+ home/user/important`

Exclude (-)

Action: Exclude matching files and directories from the backup.

- Files matching this pattern will be skipped
- Borgitory will still recurse into excluded directories to check for included files
- Most commonly used action for filtering out unwanted files

Example: `- *.tmp`

Exclude with No Recursion (!)

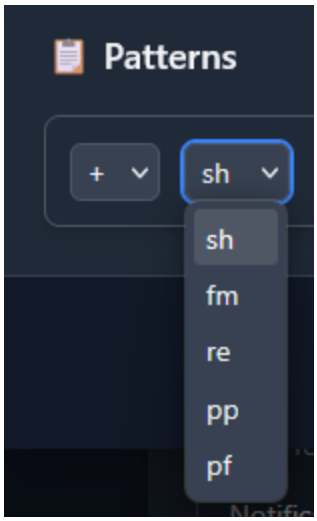
Action: Exclude matching directories and do not recurse into them.

- Completely skips the directory and all its contents
- More efficient than regular exclude for large directories you never want to backup
- Use for system directories, caches, or other large directories to ignore entirely

Example: `! home/user/.cache`

Pattern Styles

Borgitory supports multiple pattern matching styles, each with different syntax and capabilities:



Shell Style (sh)

Default style for most pattern operations. Similar to shell glob patterns with enhanced directory matching.

Features: - * matches any number of characters (except path separators) - ? matches any single character - ** matches zero or more directory levels - `[...]` matches character ranges - `[!...]` matches characters not in range

Examples: - `*.log` - All log files - `home/*/Documents` - Documents folder in any user's home - `**/*.tmp` - All .tmp files in any subdirectory - `home/user/**` - Everything under home/user

Fnmatch Style (fm)

Traditional shell pattern matching without the ** directory wildcard.

Features: - * matches any characters including path separators - ? matches any single character - `[...]` and `[!...]` for character matching - No special ** syntax

Examples: - `*.o` - All .o files - `home/*/junk` - junk directories in user homes - `cache/*` - Everything directly under cache directories

Regular Expression (re)

Full regular expression support for complex pattern matching.

Features: - Complete regex syntax support - Path separators normalized to forward slashes - Substring matching (anchor with ^ and $ as needed) - Most powerful but complex option

Examples: - `^home/[^/]+\.tmp/` - Directories ending in .tmp under home - `\.(jpg|png|gif)$` - Image files with specific extensions - `^.*/(cache|tmp)/` - Any cache or tmp directory

Warning: Regular expressions can be complex and potentially slow. Use simpler styles when possible.

Path Prefix (pp)

Matches entire directory trees by path prefix.

Features: - Matches the specified path and everything under it - Very efficient for large directory exclusions - Simple substring matching at path level

Examples: - `pp:home/user/Downloads` - Downloads directory and all contents - `pp:var/cache` - System cache directory tree - `pp:tmp` - All temporary directories

Path Full-match (pf)

Matches exact file paths only.

Features: - Extremely efficient O(1) hashtable lookup - Must specify complete, exact paths - No wildcards or variables allowed - Ignores pattern order due to hashtable implementation

Examples: - `pf:home/user/secret.txt` - Exact file match - `pf:etc/passwd` - Specific system file - `pf:var/log/system.log` - Exact log file

Pattern Processing Order

Critical: Patterns are processed in the order they appear in your configuration. The order determines which patterns take precedence when there are conflicts.

Processing Rules

1. **Top to Bottom**: Patterns are evaluated from first to last in your list
2. **First Match Wins**: The first pattern that matches a file determines its fate
3. **Include Overrides**: Later include patterns can override earlier exclude patterns
4. **Efficiency**: Place more specific patterns before general ones

Example Pattern Order

```
+ home/user/important/**     # Include important files first
- home/user/**/*.tmp         # Exclude temp files
- home/user/.cache/**        # Exclude cache
+ home/user/**               # Include everything else in user home
! var/cache/**               # Never recurse into system cache
```

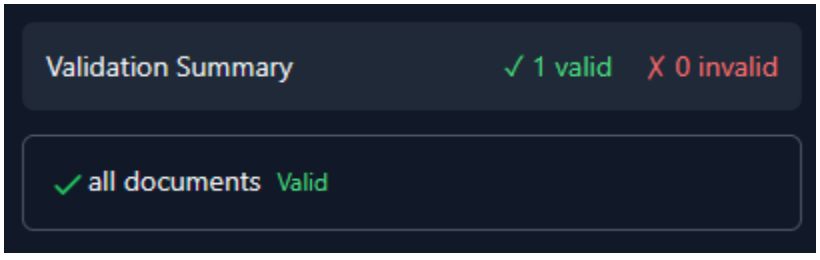
In this example: - Important files are included even if they match later exclude patterns - Temp files are excluded from user directories - Cache directories are excluded - All other user files are included - System cache is completely skipped

Best Practices

1. **Specific Before General**: Place specific include patterns before general exclude patterns
2. **Performance**: Use `!` (no recursion) for large directories you never want
3. **Testing**: Use the validation feature to test your patterns
4. **Documentation**: Add descriptive names to complex patterns

Pattern Validation

Borgitory provides real-time pattern validation to help you verify your configuration:



The validation system:

- **Syntax Check**: Verifies pattern syntax is correct
- **Preview**: Shows which files would match your patterns
- **Conflict Detection**: Identifies potential pattern conflicts
- **Performance Warnings**: Alerts about potentially slow patterns

Common Pattern Examples

System Files and Caches

```
! /proc                    # Skip system process directory
! /sys                     # Skip system directory
! /dev                     # Skip device directory
- **/.cache/**             # Exclude all cache directories
- **/node_modules/**       # Exclude Node.js dependencies
- **/__pycache__/**        # Exclude Python cache
```

User Data

```
+ home/*/Documents/**      # Include all user documents
+ home/*/Pictures/**       # Include all user pictures
- home/*/Downloads/**      # Exclude downloads
- home/*/.local/share/Trash/** # Exclude trash
```

Development Projects

```
+ home/user/projects/**/*.py    # Include Python source files
+ home/user/projects/**/*.js    # Include JavaScript files
+ home/user/projects/**/README* # Include documentation
- home/user/projects/**/build/** # Exclude build directories
- home/user/projects/**/.git/** # Exclude git repositories
```

Media Files

```
+ **/*.jpg                 # Include JPEG images
+ **/*.png                 # Include PNG images
+ **/*.mp4                 # Include MP4 videos
- **/*.tmp                 # Exclude temporary files
- **/Thumbs.db             # Exclude Windows thumbnails
```

Troubleshooting Patterns

Pattern Not Matching

1. **Check Order**: Ensure your pattern isn't overridden by an earlier pattern
2. **Verify Syntax**: Use the validation feature to check pattern syntax
3. **Test Style**: Try different pattern styles (sh vs fm vs re)
4. **Path Format**: Ensure paths use forward slashes, even on Windows

Performance Issues

1. **Use Path Prefix**: Replace complex patterns with `pp:` when possible
2. **Avoid Complex Regex**: Simplify regular expressions or use shell patterns
3. **No Recursion**: Use `!` instead of `-` for large excluded directories
4. **Order Optimization**: Place frequently matching patterns first

Unexpected Inclusions

1. **Check Include Patterns**: Later `+` patterns override earlier `-` patterns
2. **Verify Wildcards**: Ensure `*` and `**` behave as expected
3. **Test Incrementally**: Add patterns one at a time to isolate issues

Advanced Pattern Techniques

Conditional Includes

Include specific files only from certain directories:

```
+ home/*/projects/**/*.py    # Python files in projects
+ home/*/work/**/*.doc       # Documents in work folders
- home/**                    # Exclude everything else in home
```

Layered Exclusions

Create multiple layers of exclusions with exceptions:

```
+ **/important/**            # Always include important directories
- **/.git/**                 # Exclude git repositories
- **/node_modules/**         # Exclude dependencies
+ **/*.config                # But include config files
+ **                         # Include everything else
```

See Also

- *Installation Guide* - Setting up Borgitory
- *How to Set Up Automated Backup Workflows* - Creating scheduled backups
- *Troubleshooting Guide* - Common issues and solutions
- BorgBackup Pattern Documentation - Official BorgBackup pattern reference

Tip

Start with simple patterns and gradually add complexity. Use the validation feature frequently to ensure your patterns work as expected before running actual backups.

5.4.5 How to Set Up Automated Backup Workflows

This guide demonstrates how to create comprehensive automated backup workflows that combine scheduling, pruning, cloud sync, and notifications for a complete backup solution.

Overview

An automated backup workflow in Borgitory typically includes:

- **Scheduled Backups** - Regular backup execution via cron
- **Pruning Policies** - Automatic cleanup of old archives
- **Cloud Synchronization** - Offsite backup storage
- **Notifications** - Alerts for success/failure
- **Monitoring** - Health checks and reporting

This guide will walk through setting up several common workflow patterns.

Workflow 1: Daily Document Backup

This workflow backs up important documents daily with cloud sync and notifications.

Step 1: Create Repository

1. **Add Document Repository**
   - Go to Repositories → Add Repository
   - Configure repository settings:

```
Name: Documents Backup
Path: /mnt/repos/documents
Passphrase: [secure-passphrase]
```

   - Test connection and save

Step 2: Create Pruning Policy

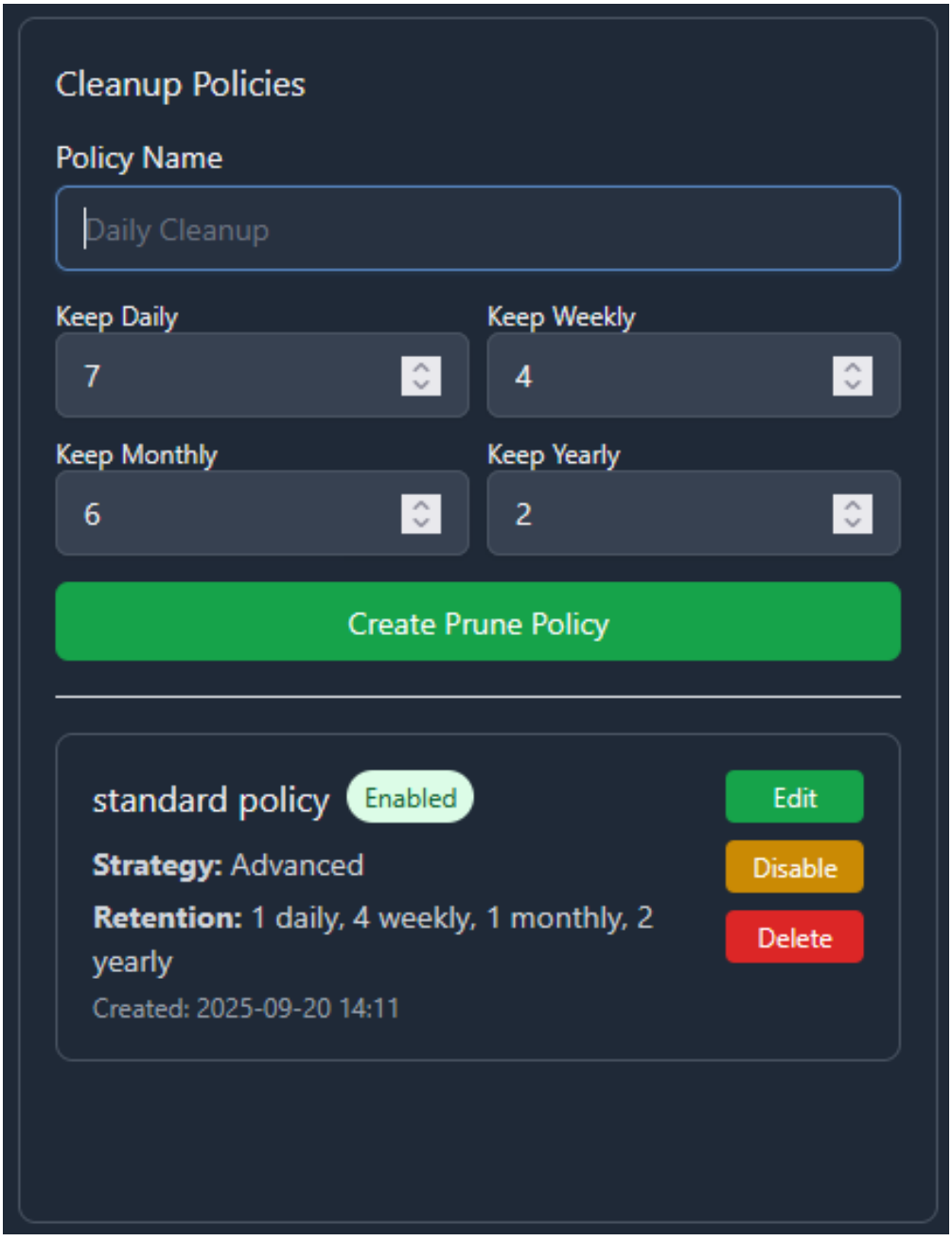
1. **Add Prune Policy**
   - Go to Archive Pruning
   - Create a new pruning policy with custom retention settings

Step 3: Configure Cloud Sync

1. **Add S3 Configuration**
   - Go to Cloud Sync → Add Configuration
   - Configure S3 settings:

```
Name: Documents S3 Backup
Provider: s3
Access Key ID: [your-access-key]
Secret Access Key: [your-secret-key]
Bucket Name: my-documents-backup
```

(continues on next page)


Cleanup Policies
Policy Name
Daily Cleanup
Keep Daily
7
Keep Weekly
4
Keep Monthly
6
Keep Yearly
2
Create Prune Policy
standard policy
Enabled
Edit
Strategy: Advanced
Disable
Retention: 1 daily, 4 weekly, 1 monthly, 2 yearly
Delete
Created: 2025-09-20 14:11

(continued from previous page)

```
Region: us-east-1
Path Prefix: documents/
```

- Test connection and save

Step 4: Set Up Notifications

1. **Configure Pushover** (see *How to Set Up Pushover Notifications*)
   - Add Pushover notification configuration
   - Test notification delivery

Step 5: Create Automated Schedule

1. **Add Backup Schedule**
   - Go to Schedules → Add Schedule
   - Choose your scheduling method:

   Option 1: Use Predefined Schedule

   Select from common scheduling options using the dropdown menu:

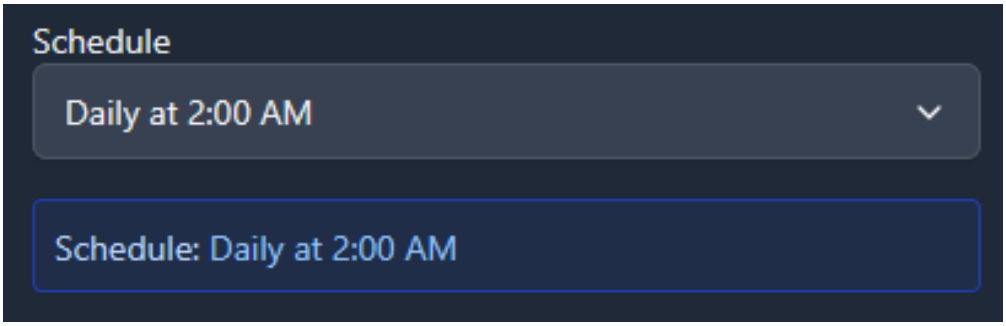


Fig. 1: Select from predefined scheduling options for common backup frequencies

   Option 2: Create Custom Cron Expression

   For more specific timing requirements, choose "Custom (cron expression)" and enter your own cron pattern:

   For detailed information about cron expression syntax and advanced scheduling options, see the APScheduler Cron Trigger documentation.

   - Configure complete workflow:

```
Name: Daily Documents Backup
Repository: Documents Backup
Source Path: /mnt/backup/sources/documents
Cron Expression: 0 2 * * *  (daily at 2 AM)
Enabled: ✓

Archive Settings:
Archive Name: documents-{now:%Y%m%d-%H%M%S}
Compression: lz4
```

(continues on next page)

Fig. 2: Create custom cron expressions for precise scheduling control

(continued from previous page)

```
Pruning:
Pruning Policy: Documents Prune
Run After Backup: ✓

Cloud Sync:
Cloud Configuration: Documents S3 Backup
Sync After Backup: ✓

Notifications:
Notification Config: Mobile Alerts
Notify on Success: ✓
Notify on Failure: ✓
Notify on Warning: ✓

Job Hooks:
Pre-Job Hooks: Database dump preparation (install postgresql-client via Package␣
↪Manager)
Post-Job Hooks: Cleanup and health checks (install curl via Package Manager)
```

2. **Save and Test**
   - Save the schedule
   - Run manually to test the complete workflow
   - Verify each step completes successfully

Workflow 2: Multi-Source System Backup

This workflow backs up multiple system directories with different schedules and prune policies.

Repository Setup

Create separate repositories for different data types:

```
System Config Repository:
- Name: System Configuration
- Path: /mnt/repos/system-config
- Purpose: /etc, system configs

User Data Repository:
- Name: User Home Directories
- Path: /mnt/repos/user-data
- Purpose: /home directories

Application Data Repository:
- Name: Application Data
- Path: /mnt/repos/app-data
- Purpose: /var/lib, databases
```

Pruning Policies

Create different prune policies:

```
System Config Prune:
- Keep Daily: 30 days
- Keep Weekly: 12 weeks
- Keep Monthly: 24 months
- Keep Yearly: 5 years

User Data Prune:
- Keep Daily: 7 days
```

(continues on next page)

(continued from previous page)

```
- Keep Weekly: 8 weeks
- Keep Monthly: 12 months
- Keep Yearly: 3 years

Application Data Prune:
- Keep Daily: 14 days
- Keep Weekly: 6 weeks
- Keep Monthly: 6 months
- Keep Yearly: 2 years
```

Schedule Configuration

Create multiple schedules with different frequencies:

System Configuration Schedule:

```
Name: System Config Backup
Repository: System Configuration
Source Path: /mnt/backup/sources/etc
Cron: 0 3 * * * (daily at 3 AM)
Archive Name: system-config-{now:%Y%m%d}
Compression: lzma (high compression for config files)
Patterns: (see :doc:`backup-patterns` for syntax details)
- /etc/shadow-
- /etc/passwd-
- *.tmp
```

User Data Schedule:

```
Name: User Data Backup
Repository: User Home Directories
Source Path: /mnt/backup/sources/home
Cron: 0 1 * * * (daily at 1 AM)
Archive Name: userdata-{now:%Y%m%d}
Compression: lz4 (fast compression for large files)
Patterns: (see :doc:`backup-patterns` for syntax details)
- .cache/
- .tmp/
- Downloads/
- .local/share/Trash/
```

Application Data Schedule:

```
Name: Application Data Backup
Repository: Application Data
Source Path: /mnt/backup/sources/var-lib
Cron: 0 4 * * * (daily at 4 AM)
Archive Name: appdata-{now:%Y%m%d}
Compression: zlib (balanced compression)
```

Monitoring and Health Checks

Workflow Health Monitoring

Create Monitoring Dashboard:

1. **Job Success Rates**
   - Monitor success/failure ratios for each workflow
   - Set up alerts for consecutive failures
   - Track backup duration trends
2. **Storage Usage Monitoring**
   - Monitor repository growth rates
   - Track cloud storage usage and costs
   - Set up alerts for rapid growth
3. **Schedule Adherence**
   - Verify schedules run on time
   - Monitor for schedule conflicts
   - Track missed backup windows

Health Check Script:

Create a health check script (`/scripts/backup_health_check.sh`):

```
#!/bin/bash
# Backup health check script

BORGITORY_API="http://localhost:8000/api"

# Check recent job status
recent_jobs=$(curl -s "$BORGITORY_API/jobs?limit=10&status=failed")

# Check repository accessibility
repositories=$(curl -s "$BORGITORY_API/repositories")

# Check cloud sync status
cloud_configs=$(curl -s "$BORGITORY_API/cloud-sync/configs")

# Generate health report
echo "Backup Health Report - $(date)"
echo "================================"

# Add health check logic here
# Send alerts if issues detected
```

Performance Optimization

Workflow Performance Tips

1. **Schedule Distribution**
   - Spread backup schedules across time

- Avoid overlapping resource-intensive operations
- Consider system load patterns

2. **Compression Strategy**
   - Use lz4 for frequently changing data
   - Use lzma for archival data
   - Use zlib for balanced performance
3. **Exclude Patterns**
   - Exclude temporary files and caches
   - Exclude large media files if not critical
   - Use specific patterns to reduce scan time
4. **Resource Management**
   - Limit concurrent backup operations
   - Monitor disk I/O during backups
   - Consider network bandwidth for cloud sync

Troubleshooting Workflows

Common Issues and Solutions

Schedule Conflicts:

```
Problem: Multiple schedules running simultaneously
Solution: Stagger schedule times, monitor resource usage
```

Cloud Sync Failures:

```
Problem: Network timeouts during large uploads
Solution: Implement retry logic, use bandwidth limiting
```

Storage Space Issues:

```
Problem: Repository storage filling up
Solution: Adjust pruning policies, monitor growth trends
```

Notification Spam:

```
Problem: Too many success notifications
Solution: Configure notifications for failures only on frequent schedules
```

Best Practices

Workflow Design Principles

1. **Start Simple** - Begin with basic workflows and add complexity gradually
2. **Test Thoroughly** - Test each component before combining into workflows
3. **Monitor Actively** - Set up monitoring and alerting for all workflows
4. **Document Everything** - Document workflow purposes and configurations

5. **Regular Review** - Periodically review and optimize workflows

Security Considerations

1. **Credential Management** - Use secure storage for cloud credentials
2. **Access Control** - Limit access to backup repositories
3. **Encryption** - Use strong passphrases for repositories
4. **Network Security** - Secure network connections for cloud sync

Next Steps

- Review *Monitoring Backup Health* for comprehensive monitoring
- Set up *Performance Optimization Guide* for better workflow performance
- Configure *Multi-Cloud Sync Setup Guide* for additional redundancy
- Explore *Troubleshooting Guide* for workflow-specific issues

With automated workflows configured, your backup infrastructure will run reliably with minimal manual intervention, providing comprehensive protection for your data with proper monitoring and alerting.

5.4.6 Backup Strategies Guide

This guide covers different backup strategies and best practices for various use cases.

Coming Soon

This guide will cover:

- 3-2-1 backup strategy implementation
- Incremental vs. full backup strategies
- Data classification and prioritization
- Backup scheduling optimization
- Recovery planning

For now, please refer to the *How to Set Up Automated Backup Workflows* guide for workflow examples.

5.4.7 Exclude Patterns Guide

> **Note**
>
> This guide has been superseded by the comprehensive *Backup Patterns Guide* guide, which covers all pattern types including include and exclude patterns with detailed examples and best practices.

This guide explains how to use exclude patterns effectively to optimize your backups and avoid backing up unnecessary files.

Quick Start

For complete pattern documentation, see *Backup Patterns Guide*. This page provides a quick reference for common exclude patterns.

Common Exclude Patterns

System and Cache Files

```
- **/.cache/**                # All cache directories
- **/node_modules/**          # Node.js dependencies
- **/__pycache__/**           # Python cache
- **/.git/**                  # Git repositories
- **/Thumbs.db                # Windows thumbnails
- **/.DS_Store                # macOS metadata
```

Temporary Files

```
- **/*.tmp                    # Temporary files
- **/*.temp                   # Temporary files
- **/*~                       # Backup files
- **/*.swp                    # Vim swap files
- **/*.log                    # Log files
```

Media and Downloads

```
- **/Downloads/**             # Download directories
- **/*.iso                    # ISO images
- **/*.dmg                    # macOS disk images
- **/Trash/**                 # Trash directories
```

For complete pattern syntax, advanced techniques, and detailed examples, see the *Backup Patterns Guide* guide.

5.4.8 Job Hooks System Documentation

Overview

The Job Hooks System allows you to execute custom commands before and after backup jobs. This powerful feature enables you to:

- Prepare systems before backups (database dumps, service stops, etc.)
- Perform cleanup operations after backups
- Run health checks and validations
- Send custom notifications or alerts
- Integrate with external monitoring systems

Note

Many hook scripts require additional command-line tools (like `curl`, `jq`, `postgresql-client`, etc.). Use the *Package Manager for Pre/Post Hooks* to install these tools directly from the Borgitory interface.

Hook Types

Pre-Job Hooks

Pre-job hooks execute **before** the backup process begins. They are ideal for:

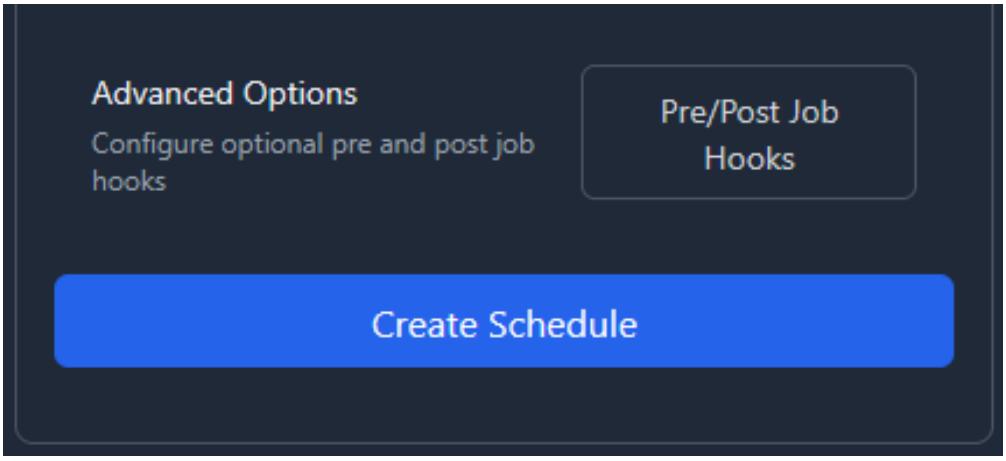
- Creating database dumps
- Stopping services that lock files
- Mounting network drives
- Validating system prerequisites
- Creating temporary directories

Post-Job Hooks

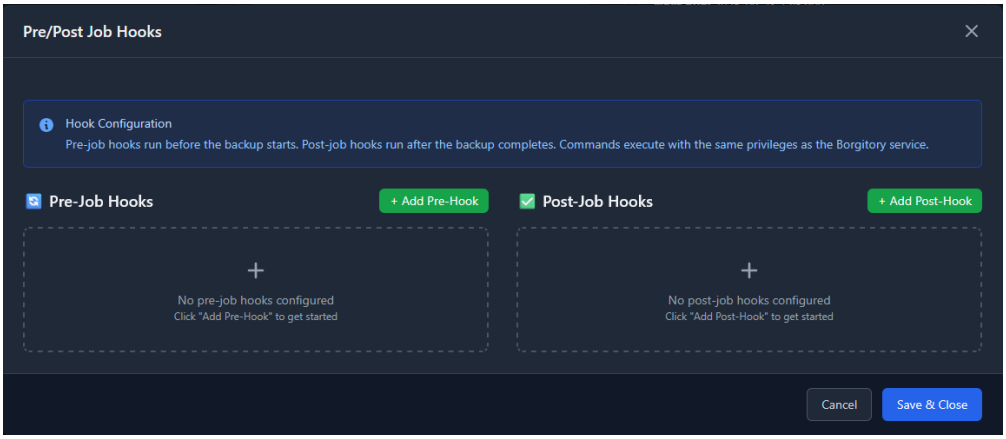
Post-job hooks execute **after** the backup process completes. They are useful for:

- Cleaning up temporary files
- Restarting services
- Running health checks
- Sending completion notifications
- Uploading logs to external systems

Adding Hooks to a Schedule



1. **Navigate to Schedule Creation/Editing**
   - Go to the Schedules page
   - Click “Create Schedule” or edit an existing schedule
2. **Open the Hooks Configuration Modal**
   - Click the “Configure Hooks” button in the schedule form
   - This opens the hooks configuration modal
3. **Add Pre-Job Hooks**


Pre/Post Job Hooks
Hook Configuration
Pre-job hooks run before the backup starts. Post-job hooks run after the backup completes. Commands execute with the same privileges as the Borgitory service.
Pre-Job Hooks
+ Add Pre-Hook
Post-Job Hooks
+ Add Post-Hook
No pre-job hooks configured
Click "Add Pre-Hook" to get started
No post-job hooks configured
Click "Add Post-Hook" to get started
Cancel
Save & Close

- In the "Pre-Job Hooks" section, click "Add Pre-Hook"
- Enter a descriptive name for your hook
- Enter the command to execute
- Configure hook options (see Hook Options section)

4. **Add Post-Job Hooks**
   - In the "Post-Job Hooks" section, click "Add Post-Hook"
   - Enter a descriptive name for your hook
   - Enter the command to execute
   - Configure hook options (see Hook Options section)
5. **Save Configuration**
   - Click "Save Hooks" to apply your configuration
   - The modal will close and return you to the schedule form
   - Complete and save your schedule

Hook Options

Critical Hooks

What it does: When a hook is marked as "Critical", the entire job will fail and stop immediately if the hook fails.

Use Cases:

Database Preparation Hook

```
# Pre-hook: Create database dump
Name: "Database Dump"
Command: "pg_dump myapp > /tmp/myapp_backup.sql"
Critical: ✓ Enabled
```

Why Critical: If the database dump fails, there's no point in continuing with the file backup since the database state won't be captured.

System Prerequisites Check

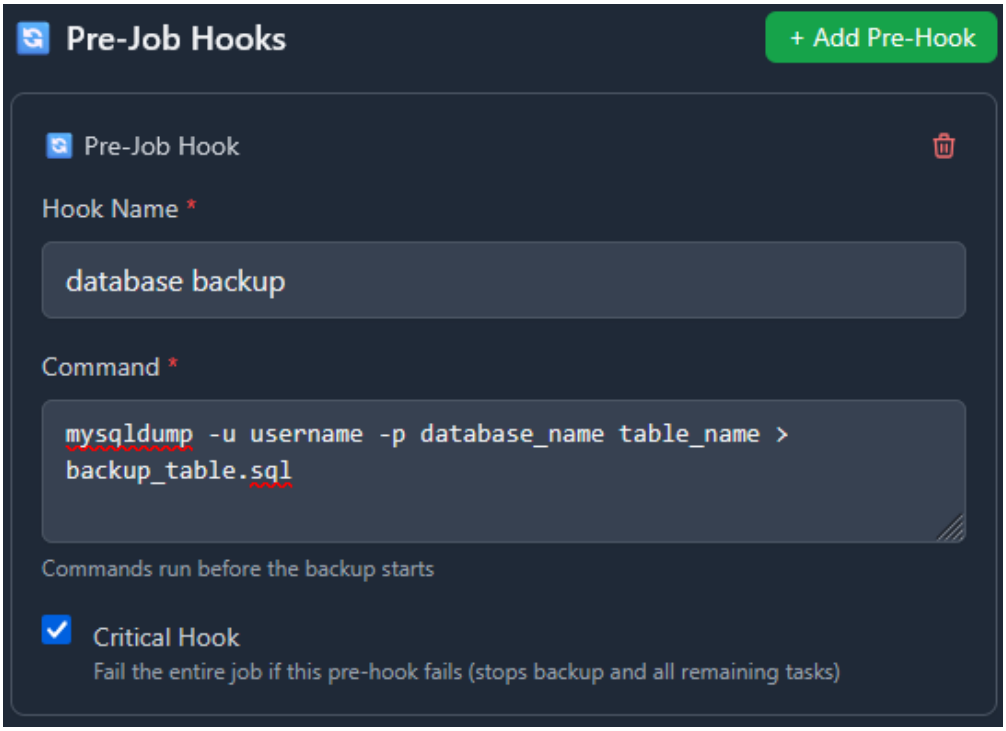
```
# Pre-hook: Check disk space
Name: "Disk Space Check"
Command: "df /backup | awk 'NR==2 {if ($4 < 1000000) exit 1}'"
Critical: ✓ Enabled
```

Why Critical: If there's insufficient disk space, the backup will fail anyway, so it's better to fail early.

Service Dependency

```
# Pre-hook: Stop application
Name: "Stop Web Service"
Command: "systemctl stop nginx && sleep 5"
Critical: ✓ Enabled
```

Why Critical: If the service can't be stopped cleanly, files may be in an inconsistent state.


Pre-Job Hooks
+ Add Pre-Hook
Pre-Job Hook
Hook Name *
database backup
Command *
mysqldump -u username -p database_name table_name > backup_table.sql
Commands run before the backup starts
Critical Hook
Fail the entire job if this pre-hook fails (stops backup and all remaining tasks)

Run on Job Failure (Post-Hooks Only)

What it does: Post-hooks with this option enabled will execute even if the backup job or a critical pre-hook failed.

Use Cases:

Health Check Notifications

```
# Post-hook: Always send health check
Name: "Health Check Ping"
Command: "curl -X POST https://healthcheck.io/ping/abc123"
Run Even If Job Failed: ✓ Enabled
```

Why Always Run: External monitoring systems need to know about both successful and failed backups.

Cleanup Operations

```
# Post-hook: Clean temporary files
Name: "Cleanup Temp Files"
Command: "rm -rf /tmp/backup_staging/*"
Run Even If Job Failed: ✓ Enabled
```

Why Always Run: Temporary files should be cleaned up regardless of backup success to prevent disk space issues.

Service Restart

```
# Post-hook: Restart services
Name: "Restart Application"
Command: "systemctl start nginx && systemctl start mysql"
Run Even If Job Failed: ✓ Enabled
```

Why Always Run: Services should be restarted even if the backup failed to ensure system availability.

Hook Execution Flow

Normal Execution (All Successful)

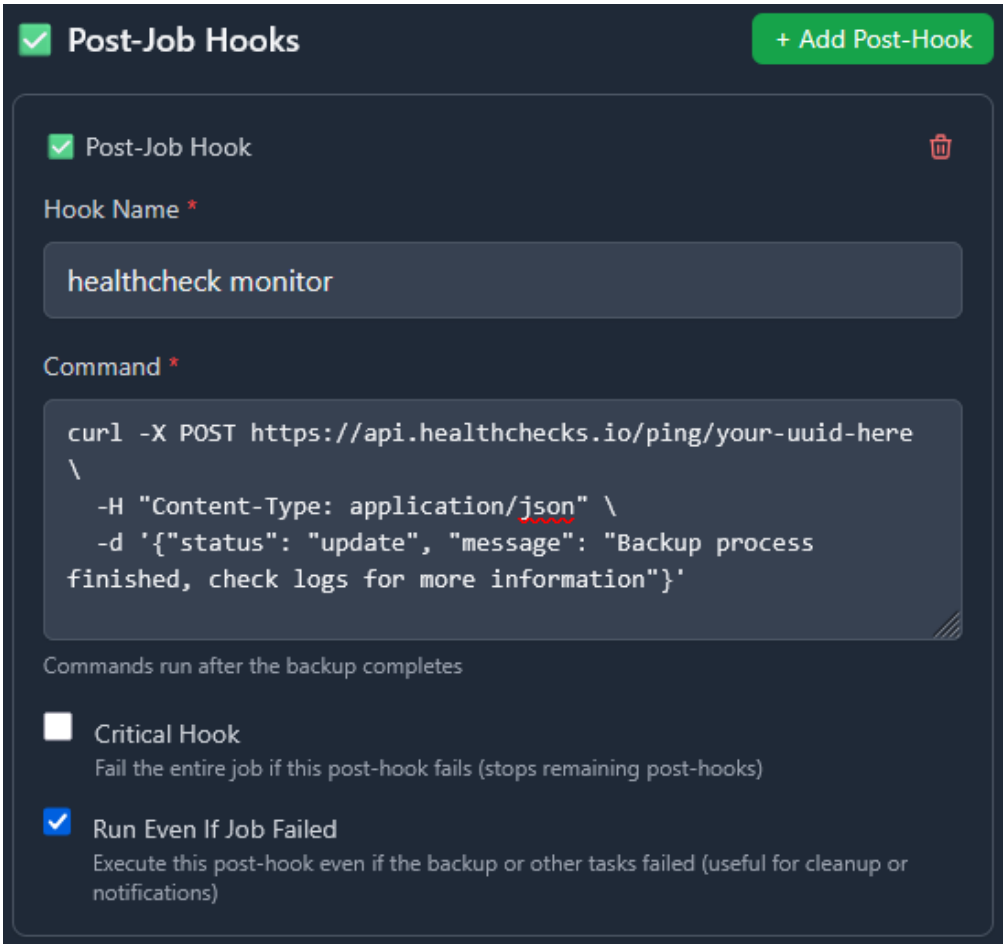
```
1. Pre-Hook 1 (Database Dump)        →  Success
2. Pre-Hook 2 (Stop Services)        →  Success
3. Backup Task                        →  Success
4. Prune Task                         →  Success
5. Post-Hook 1 (Restart Services)    →  Success
6. Post-Hook 2 (Health Check)        →  Success
7. Notification                       →  Success

Result:  Job Completed Successfully
```

Critical Pre-Hook Failure

```
1. Pre-Hook 1 (Database Dump)        →  Failed (Critical)
2. Pre-Hook 2 (Stop Services)        →   Skipped
3. Backup Task                        →   Skipped
4. Prune Task                         →   Skipped
5. Post-Hook 1 (Restart Services)    →   Skipped
6. Post-Hook 2 (Health Check)        →  Success (Run on Failure)
7. Notification                       →  Success
```

(continues on next page)


Post-Job Hooks
+ Add Post-Hook
Post-Job Hook
Hook Name *
healthcheck monitor
Command *
curl -X POST https://api.healthchecks.io/ping/your-uuid-here
\
-H "Content-Type: application/json" \
-d '{"status": "update", "message": "Backup process
finished, check logs for more information"}'
Commands run after the backup completes
Critical Hook
Fail the entire job if this post-hook fails (stops remaining post-hooks)
Run Even If Job Failed
Execute this post-hook even if the backup or other tasks failed (useful for cleanup or
notifications)

(continued from previous page)

```
Result:  Job Failed - Critical Hook Error
```

Backup Task Failure

```
1. Pre-Hook 1 (Database Dump)        →  Success
2. Pre-Hook 2 (Stop Services)        →  Success
3. Backup Task                        →  Failed (Critical)
4. Prune Task                         →   Skipped
5. Post-Hook 1 (Restart Services)    →  Success (Run on Failure)
6. Post-Hook 2 (Health Check)        →  Success (Run on Failure)
7. Notification                       →  Success

Result:  Job Failed - Backup Error
```

Non-Critical Hook Failure

```
1. Pre-Hook 1 (Database Dump)        →  Success
2. Pre-Hook 2 (Log Rotation)         →  Failed (Non-Critical)
3. Backup Task                        →  Success
4. Prune Task                         →  Success
5. Post-Hook 1 (Restart Services)    →  Success
6. Post-Hook 2 (Health Check)        →  Success
7. Notification                       →  Success

Result:   Job Completed with Warnings
```

Task Status Indicators

When viewing job history, tasks will show different statuses based on execution results:

- **Completed:** Task executed successfully
- **Failed:** Task executed but failed
- **Skipped:** Task was not executed due to an earlier critical failure
- **Pending:** Task has not yet been executed (job still running)

Skipped Task Behavior

When a critical hook or backup task fails, all subsequent tasks are automatically marked as "Skipped" with an explanatory message:

- **Critical Hook Failure:** "Task skipped due to critical hook failure"
- **Critical Task Failure:** "Task skipped due to critical task failure"
- **Critical Task Exception:** "Task skipped due to critical task exception"

Environment Variables

Hook scripts automatically receive environment variables with job context:

```
# Available in all hook scripts
BORGITORY_REPOSITORY_ID="123"     # ID of the repository being backed up
BORGITORY_TASK_INDEX="2"          # Position of this hook in the job sequence
BORGITORY_JOB_TYPE="scheduled"    # Type of job (scheduled, manual)
```

Example hook using environment variables:

```
#!/bin/bash
# Pre-hook: Context-aware database dump

DB_NAME="app_repo_${BORGITORY_REPOSITORY_ID}"
BACKUP_FILE="/tmp/dump_${BORGITORY_REPOSITORY_ID}_$(date +%Y%m%d_%H%M%S).sql"

echo "Creating dump for repository ${BORGITORY_REPOSITORY_ID}"
pg_dump "$DB_NAME" > "$BACKUP_FILE"

if [ $? -eq 0 ]; then
    echo "Database dump created: $BACKUP_FILE"
else
    echo "Failed to create database dump for repository ${BORGITORY_REPOSITORY_ID}"
    exit 1
fi
```

Notification Messages

The notification system provides detailed information about hook failures:

Successful Job

```
 Backup Job Completed Successfully

Backup job for 'MyRepository' completed successfully.

Tasks Completed: 7, Total: 7
Job ID: job-abc-123
```

Critical Hook Failure

```
 Backup Job Failed - Critical Hook Error

Backup job for 'MyRepository' failed due to critical hook failure.

Failed Hook: Database Dump
Tasks Completed: 0, Skipped: 5, Total: 7
Job ID: job-abc-123
```

Job with Warnings

```
 Backup Job Completed with Warnings

Backup job for 'MyRepository' completed but some tasks failed.

Failed Tasks: hook
Tasks Completed: 6, Skipped: 0, Total: 7
Job ID: job-abc-123

.. figure:: /_static/how-to/job-hooks/failed_prehook.png
   :alt: How to open the failed pre-hook
   :width: 80%
   :align: center
```

Common Hook Examples

Database Backup Hooks

PostgreSQL Dump (Pre-Hook)

```
# Name: "PostgreSQL Database Dump"
# Critical: ✓ Enabled

#!/bin/bash
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
DUMP_FILE="/backup/staging/postgres_${TIMESTAMP}.sql"

pg_dump -h localhost -U backup_user myapp_db > "$DUMP_FILE"

if [ $? -eq 0 ]; then
    echo "Database dump created: $DUMP_FILE"
else
    echo "Failed to create PostgreSQL dump"
    exit 1
fi
```

MySQL Dump (Pre-Hook)

```
# Name: "MySQL Database Dump"
# Critical: ✓ Enabled

#!/bin/bash
mysqldump -u backup_user -p$MYSQL_PASSWORD myapp_db > /backup/staging/mysql_dump.sql

if [ $? -eq 0 ]; then
    echo "MySQL dump completed successfully"
else
    echo "MySQL dump failed"
    exit 1
fi
```

Service Management Hooks

Stop Services (Pre-Hook)

```
# Name: "Stop Application Services"
# Critical: ✓ Enabled

#!/bin/bash
echo "Stopping application services..."

systemctl stop nginx
systemctl stop php-fpm
systemctl stop redis

# Wait for services to fully stop
sleep 10

echo "Services stopped successfully"
```

Restart Services (Post-Hook)

```
# Name: "Restart Application Services"
# Run Even If Job Failed: ✓ Enabled

#!/bin/bash
echo "Restarting application services..."

systemctl start redis
systemctl start php-fpm
systemctl start nginx

# Verify services are running
if systemctl is-active --quiet nginx && systemctl is-active --quiet php-fpm; then
    echo "Services restarted successfully"
else
    echo "Warning: Some services may not have started properly"
    exit 1
fi
```

Health Check and Monitoring Hooks

System Health Check (Post-Hook)

```
# Name: "System Health Check"
# Run Even If Job Failed: ✓ Enabled

#!/bin/bash

# Check disk space
DISK_USAGE=$(df /backup | awk 'NR==2 {print $5}' | sed 's/%//')
if [ "$DISK_USAGE" -gt 90 ]; then
    echo "Warning: Backup disk usage is ${DISK_USAGE}%"
```

(continues on next page)

(continued from previous page)

```
fi

# Check system load
LOAD=$(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}' | sed 's/,//')
echo "Current system load: $LOAD"

# Ping monitoring service
curl -X POST "https://healthcheck.io/ping/your-uuid-here" \
     -d "Backup completed. Disk: ${DISK_USAGE}%, Load: ${LOAD}"
```

External Service Integration (Post-Hook)

```
# Name: "Update Monitoring Dashboard"
# Run Even If Job Failed: ✓ Enabled

#!/bin/bash

# Determine backup status from environment or job context
if [ "$BORGITORY_JOB_TYPE" = "scheduled" ]; then
    STATUS="scheduled_backup_complete"
else
    STATUS="manual_backup_complete"
fi

# Send to monitoring API
curl -X POST "https://monitoring.company.com/api/events" \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer $MONITORING_TOKEN" \
     -d "{
       \"event\": \"$STATUS\",
       \"repository_id\": \"$BORGITORY_REPOSITORY_ID\",
       \"timestamp\": \"$(date -Iseconds)\"
     }"
```

File System Preparation Hooks

Mount Network Storage (Pre-Hook)

```
# Name: "Mount Network Backup Storage"
# Critical: ✓ Enabled

#!/bin/bash

MOUNT_POINT="/mnt/backup_storage"
NFS_SERVER="backup-server.company.com:/exports/backups"

# Check if already mounted
if mountpoint -q "$MOUNT_POINT"; then
    echo "Network storage already mounted"
    exit 0
fi
```

(continues on next page)

(continued from previous page)

```
# Create mount point if it doesn't exist
mkdir -p "$MOUNT_POINT"

# Mount the NFS share
mount -t nfs "$NFS_SERVER" "$MOUNT_POINT"

if [ $? -eq 0 ]; then
    echo "Network storage mounted successfully"
else
    echo "Failed to mount network storage"
    exit 1
fi
```

Cleanup Temporary Files (Post-Hook)

```
# Name: "Cleanup Staging Area"
# Run Even If Job Failed: ✓ Enabled

#!/bin/bash

STAGING_DIR="/backup/staging"

echo "Cleaning up staging directory: $STAGING_DIR"

# Remove files older than 1 day from staging
find "$STAGING_DIR" -type f -mtime +1 -delete

# Remove empty directories
find "$STAGING_DIR" -type d -empty -delete

echo "Staging cleanup completed"
```

Best Practices

Hook Design Guidelines

1. **Make Hooks Idempotent**

   Hooks should be safe to run multiple times without causing issues.

2. **Use Descriptive Names**

   Choose clear, descriptive names that explain what the hook does.

3. **Handle Errors Gracefully**

   Use proper exit codes and error messages for debugging.

4. **Log Important Information**

   Include relevant output for troubleshooting and monitoring.

5. **Test Hooks Independently**

   Test your hook scripts manually before adding them to schedules.

Critical vs Non-Critical Decision Matrix

Mark a hook as **Critical** if:

- The backup is meaningless without the hook's success
- Continuing would cause data corruption or inconsistency
- System prerequisites are not met
- The hook failure indicates a serious system problem

Mark a hook as **Non-Critical** if:

- The hook is for optimization or convenience
- Backup can still be valuable even if the hook fails
- The hook is for monitoring or reporting
- Failure is recoverable or acceptable

Run on Job Failure Decision Matrix

Enable **Run Even If Job Failed** for post-hooks that:

- Restore system state (restart services, unmount drives)
- Perform cleanup operations
- Send monitoring/health check pings
- Handle failure notifications
- Reset system configuration

Security Considerations

Hook Script Security

1. **Use Absolute Paths**

   Always use full paths to executables to prevent PATH manipulation attacks.

2. **Validate Input**

   If your hooks accept parameters, validate them thoroughly.

3. **Limit Permissions**

   Run hooks with the minimum required permissions.

4. **Secure Credentials**

   Store sensitive information in environment variables or secure credential stores.

5. **Log Security Events**

   Log authentication attempts and access to sensitive resources.

File System Permissions

Ensure hook scripts have appropriate permissions:

```
# Make hook executable by owner only
chmod 700 /path/to/hook-script.sh

# Set appropriate ownership
chown borgitory:borgitory /path/to/hook-script.sh
```

Troubleshooting

Common Issues and Solutions

Hook Not Executing

Symptoms: Hook shows as "Failed" immediately without output

Possible Causes: - Script file doesn't exist or isn't executable - Incorrect shebang line (#!/bin/bash) - Permission denied

Solutions: - Verify script path and permissions: `ls -la /path/to/script.sh` - Check shebang line is correct - Ensure script is executable: `chmod +x /path/to/script.sh`

Hook Times Out

Symptoms: Hook shows as "Failed" after exactly 5 minutes (default timeout)

Solutions: - Increase hook timeout in configuration - Optimize script performance - Add progress logging to identify bottlenecks

Environment Variables Not Available

Symptoms: Hook script can't access BORGITORY_* variables

Solutions: - Use `env | grep BORGITORY` in hook to debug available variables - Ensure script uses correct variable names (case sensitive) - Check if custom shell environment affects variable access

Database Connection Failures

Symptoms: Database hooks fail with connection errors

Solutions: - Verify database credentials and connectivity - Check if database service is running - Test connection manually: `psql -h host -U user -d database -c "SELECT 1;"` - Consider connection timeouts and retry logic

Service Start/Stop Issues

Symptoms: Service management hooks fail inconsistently

Solutions: - Add delays after service operations: `sleep 5` - Check service status before operations: `systemctl is-active service` - Use proper service dependencies and ordering - Consider using service-specific health checks

Debugging Hook Execution

Enable Detailed Logging

Add debugging output to your hooks:

```
#!/bin/bash
set -x  # Enable debug output
set -e  # Exit on any error
```

(continues on next page)

(continued from previous page)

```
echo "Hook starting at $(date)"
echo "Environment: $(env | grep BORGITORY)"

# Your hook logic here

echo "Hook completed at $(date)"
```

Test Hooks Manually

Run hooks outside of Borgitory to test:

```
# Set up environment variables manually
export BORGITORY_REPOSITORY_ID="123"
export BORGITORY_TASK_INDEX="1"
export BORGITORY_JOB_TYPE="manual"

# Run your hook script
/path/to/your/hook-script.sh
```

Monitor System Resources

Check system resources during hook execution:

```
# Monitor disk space
df -h

# Monitor memory usage
free -h

# Monitor running processes
ps aux | grep your-hook-process
```

Advanced Configuration

Custom Shell Configuration

Hooks can specify custom shell interpreters:

```
# Python hook
#!/usr/bin/env python3
import os
import subprocess

repo_id = os.environ.get('BORGITORY_REPOSITORY_ID')
print(f"Processing repository {repo_id}")
```

```
# PowerShell hook (Windows)
#!/usr/bin/env pwsh
$RepoId = $env:BORGITORY_REPOSITORY_ID
Write-Host "Processing repository $RepoId"
```

Working Directory

Hooks execute in the Borgitory application directory by default. You can change directories within your hook:

```
#!/bin/bash
cd /path/to/your/working/directory
# Hook operations here
```

Migration from Legacy Systems

If you're migrating from cron jobs or other backup systems:

From Cron Jobs

Old cron approach:

```
# Crontab entry
0 2 * * * /usr/local/bin/pre-backup.sh && /usr/local/bin/backup.sh && /usr/local/bin/
↪post-backup.sh
```

New hook approach:

1. Move `pre-backup.sh` content to a pre-hook
2. Configure Borgitory backup normally
3. Move `post-backup.sh` content to a post-hook
4. Remove cron job

From Shell Scripts

Old monolithic script:

```
#!/bin/bash
# pre-backup operations
pg_dump mydb > /tmp/dump.sql

# backup
borg create repo::backup /data

# post-backup operations
rm /tmp/dump.sql
```

New hook-based approach:

- **Pre-hook:** `pg_dump mydb > /tmp/dump.sql`
- **Backup:** Configured in Borgitory UI
- **Post-hook:** `rm /tmp/dump.sql`

API Integration

For advanced users, hooks can be managed via the Borgitory API:

Creating Hooks via API

```
# Create schedule with hooks
curl -X POST "http://borgitory/api/schedules" \
     -H "Content-Type: application/json" \
     -d '{
       "name": "Database Backup",
       "repository_id": 1,
       "pre_job_hooks": "[{\"name\":\"DB Dump\",\"command\":\"pg_dump mydb\",\"critical\
↪":true}]",
       "post_job_hooks": "[{\"name\":\"Cleanup\",\"command\":\"rm /tmp/*\",\"run_on_job_
↪failure\":true}]"
     }'
```

Monitoring Hook Status

```
# Get job status including hook results
curl "http://borgitory/api/jobs/123" | jq '.tasks[] | select(.task_type=="hook")'
```

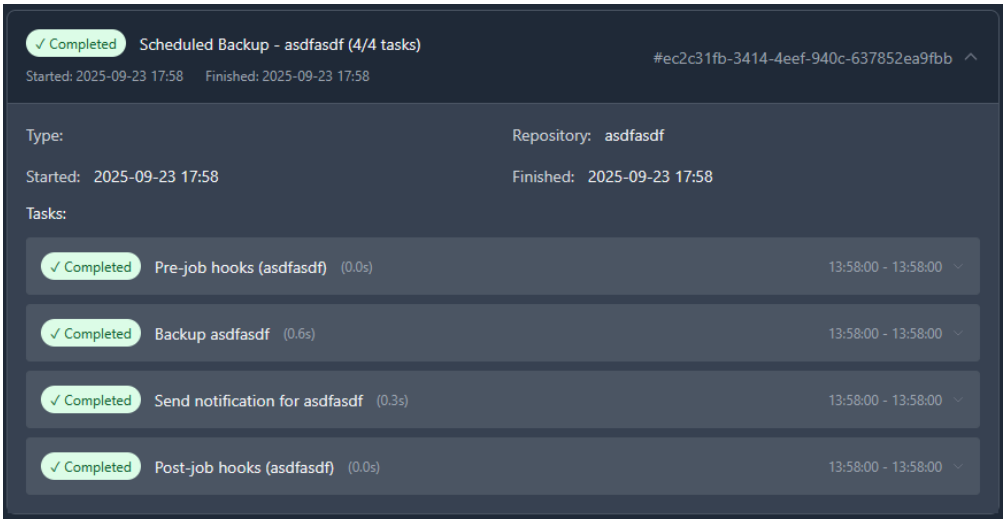
Conclusion

The Job Hooks System provides powerful automation capabilities for your backup workflows. By combining pre-hooks, post-hooks, critical failure handling, and conditional execution, you can create robust, automated backup processes that handle both success and failure scenarios gracefully.

Key benefits:

- **Automated preparation and cleanup**
- **Intelligent failure handling**
- **Integration with external systems**
- **Detailed execution tracking**
- **Flexible configuration options**

Start with simple hooks and gradually build more sophisticated automation as your needs grow. The system is designed to be both powerful for advanced users and accessible for those just getting started with backup automation.



5.4.9 How to Set Up AWS S3 Cloud Sync

This guide walks you through configuring Amazon S3 for cloud synchronization with Borgitory.

Prerequisites

Before starting, ensure you have:

- An AWS account with S3 access
- An S3 bucket created for storing backups
- AWS credentials (Access Key ID and Secret Access Key)
- Borgitory installed with Rclone support

Step 1: Create AWS Credentials

Create IAM User

1. **Log into AWS Console**
   - Navigate to the IAM service
   - Click "Users" in the left sidebar
2. **Create New User**
   - Click "Create user"
   - Enter username: `borgitory-backup-user`
   - Select "Programmatic access"
3. **Attach Permissions**
   - Choose "Attach existing policies directly"
   - Search for and select `AmazonS3FullAccess`
   - Or create a custom policy for specific bucket access
4. **Save Credentials**
   - Download the credentials CSV file
   - Note the Access Key ID and Secret Access Key
   - Store these securely - they won't be shown again

Custom S3 Policy (Recommended)

For better security, create a custom policy that only allows access to your backup bucket:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
```

(continues on next page)

(continued from previous page)

```
                "s3:GetBucketLocation"
            ],
            "Resource": "arn:aws:s3:::your-backup-bucket"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:DeleteObject"
            ],
            "Resource": "arn:aws:s3:::your-backup-bucket/*"
        }
    ]
}
```

Replace `your-backup-bucket` with your actual bucket name.

Step 2: Create S3 Bucket

Bucket Configuration

1. **Create Bucket**
   - Navigate to S3 service in AWS Console
   - Click "Create bucket"
   - Enter bucket name: `my-borgitory-backups`
   - Choose appropriate region (closer to your location for better performance)
2. **Configure Security**
   - **Block Public Access**: Keep all settings enabled (recommended)
   - **Bucket Versioning**: Enable if you want version history
   - **Server-side Encryption**: Enable with S3 managed keys (SSE-S3)
3. **Lifecycle Policies** (Optional)
   - Configure automatic transitions to cheaper storage classes
   - Set up automatic deletion of old versions

Example Lifecycle Policy

To automatically move backups to cheaper storage after 30 days:

```
{
    "Rules": [
        {
            "ID": "BorgitoryBackupLifecycle",
            "Status": "Enabled",
            "Transitions": [
                {
                    "Days": 30,
```

(continues on next page)

(continued from previous page)

```
                    "StorageClass": "STANDARD_IA"
                },
                {
                    "Days": 90,
                    "StorageClass": "GLACIER"
                },
                {
                    "Days": 365,
                    "StorageClass": "DEEP_ARCHIVE"
                }
            ]
        }
    ]
}
```

Step 3: Configure Borgitory

Add S3 Configuration

1. **Navigate to Cloud Sync**
   - Open Borgitory web interface
   - Go to Cloud Sync → Configurations
   - Click "Add Configuration"
2. **Select S3 Provider**
   - Choose "Amazon S3" from the provider dropdown
   - The S3-specific fields will appear
3. **Enter Configuration Details**

   ```
   Name: Production S3 Backup
   Provider: s3
   Access Key ID: AKIAIOSFODNN7EXAMPLE
   Secret Access Key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   Bucket Name: my-borgitory-backups
   Region: us-east-1
   Path Prefix: server-backups/
   ```

   - **Name**: Friendly name for this configuration
   - **Access Key ID**: From your AWS credentials
   - **Secret Access Key**: From your AWS credentials
   - **Bucket Name**: Your S3 bucket name
   - **Region**: AWS region where your bucket is located
   - **Path Prefix**: Optional path within bucket for organization
4. **Test Connection**
   - Click "Test Connection" button
   - Wait for green checkmark indicating success

- If test fails, verify credentials and bucket permissions

5. **Save Configuration**
   - Click "Save" to store the configuration
   - Configuration appears in the Cloud Sync list

Step 4: Test Cloud Sync

Manual Sync Test

1. **Prepare Test Repository**
   - Ensure you have a repository with at least one archive
   - Note the repository size for comparison
2. **Start Manual Sync**
   - Go to Cloud Sync → Manual Sync
   - Select your repository
   - Choose your S3 configuration
   - Click "Start Sync"
3. **Monitor Progress**
   - Watch real-time sync progress
   - Note transfer speeds and estimated completion time
   - Check for any errors or warnings
4. **Verify in AWS Console**
   - Open S3 console
   - Navigate to your bucket
   - Verify files are uploaded under the correct path prefix
   - Check file sizes match expectations

Automated Sync Setup

1. **Configure Schedule**
   - Go to Schedules
   - Edit existing schedule or create new one
   - Enable "Cloud Sync" option
   - Select your S3 configuration
2. **Test Automated Flow**
   - Run the schedule manually or wait for next execution
   - Verify backup completes successfully
   - Confirm automatic sync to S3 occurs
   - Check job history for any issues

Step 5: Optimization and Monitoring

Performance Optimization

Transfer Speed

- Choose S3 region closest to your server
- Consider using S3 Transfer Acceleration for global deployments
- Monitor bandwidth usage and adjust if needed

Storage Costs

- Use lifecycle policies to transition old backups to cheaper storage
- Monitor S3 costs in AWS billing console
- Consider using S3 Intelligent Tiering for automatic optimization

Compression Settings

- Use appropriate compression in Borgitory (lz4 for speed, lzma for size)
- Balance compression ratio vs. upload time
- Test different settings with your data

Monitoring and Alerts

CloudWatch Metrics

- Set up CloudWatch alarms for unusual S3 activity
- Monitor bucket size growth
- Track request patterns and errors

Borgitory Monitoring

- Enable push notifications for sync failures
- Monitor sync history for patterns
- Set up regular sync health checks

Cost Monitoring

- Use AWS Cost Explorer to track S3 expenses
- Set up billing alerts for unexpected costs
- Review storage class distribution regularly

Troubleshooting Common Issues

Connection Failures

Invalid Credentials

```
Error: Access Denied (403)
```

- Verify Access Key ID and Secret Access Key are correct
- Check if IAM user has necessary S3 permissions
- Ensure credentials haven't expired

Bucket Access Issues

```
Error: NoSuchBucket
```

- Verify bucket name is spelled correctly
- Check if bucket exists in the specified region
- Ensure bucket is in the same region as specified in configuration

Network Issues

```
Error: Connection timeout
```

- Check internet connectivity
- Verify firewall allows HTTPS traffic to AWS
- Consider using S3 VPC endpoints if running in AWS

Sync Failures

Insufficient Permissions

- Review IAM policy attached to user
- Ensure user has `s3:PutObject` permission
- Check bucket policies don't deny access

Storage Quota Exceeded

- Monitor S3 bucket size and costs
- Implement lifecycle policies to manage old backups
- Consider using different storage classes

Large File Issues

- Rclone automatically handles multipart uploads
- For very large repositories, consider chunking strategy
- Monitor transfer progress and retry failed uploads

Security Best Practices

Credential Management

- Use IAM roles instead of access keys when running on EC2
- Rotate access keys regularly
- Never commit credentials to version control
- Use AWS Secrets Manager for credential storage in production

Bucket Security

- Enable bucket versioning for additional protection
- Use bucket notifications to monitor access
- Enable AWS CloudTrail for audit logging

- Consider using S3 Object Lock for compliance requirements

Network Security

- Use VPC endpoints for private S3 access from EC2
- Enable S3 access logging for security monitoring
- Consider using AWS PrivateLink for enhanced security
- Implement least-privilege access policies

Next Steps

- Set up *How to Set Up Pushover Notifications* for sync alerts
- Configure *How to Set Up Automated Backup Workflows* with S3 integration
- Explore *Multi-Cloud Sync Setup Guide* for redundancy
- Review *Performance Optimization Guide* for better sync speeds

5.4.10 How to Set Up Google Cloud Storage

This guide walks you through configuring Google Cloud Storage for cloud synchronization with Borgitory.

Coming Soon

This guide will cover:

- Creating Google Cloud Storage buckets
- Setting up service account authentication
- Configuring Borgitory for GCS integration
- Optimizing storage classes and lifecycle policies
- Troubleshooting common issues

For now, please refer to the *How to Set Up AWS S3 Cloud Sync* guide for a similar cloud provider setup example.

5.4.11 How to Set Up Azure Blob Storage

This guide walks you through configuring Azure Blob Storage for cloud synchronization with Borgitory.

Coming Soon

This guide will cover:

- Creating Azure Storage accounts and containers
- Setting up authentication with access keys or SAS tokens
- Configuring Borgitory for Azure integration
- Optimizing storage tiers and lifecycle management
- Troubleshooting common issues

For now, please refer to the *How to Set Up AWS S3 Cloud Sync* guide for a similar cloud provider setup example.

5.4.12 Multi-Cloud Sync Setup Guide

This guide explains how to configure multiple cloud providers for redundant backup storage.

Coming Soon

This guide will cover:

- Multi-cloud backup strategies
- Configuring multiple cloud providers
- Setting up redundant sync schedules
- Cost optimization across providers
- Monitoring multi-cloud operations

For now, please refer to the *How to Set Up Automated Backup Workflows* guide for multi-cloud workflow examples.

5.4.13 How to Set Up Pushover Notifications

This guide walks you through setting up Pushover notifications to receive mobile alerts for backup job completions and failures.

Overview

Pushover is a simple push notification service that delivers real-time notifications to your phone, tablet, and desktop. Borgitory integrates with Pushover to send alerts about:

- Successful backup completions
- Backup failures and errors
- Cloud sync operations
- Schedule execution status

Prerequisites

Before starting, you'll need:

- A Pushover account (free tier available)
- Borgitory installed and running
- A mobile device with the Pushover app installed

Step 1: Create Pushover Account

Sign Up for Pushover

1. **Visit Pushover Website**
   - Go to https://pushover.net/
   - Click "Sign up" to create a free account
   - Verify your email address
2. **Install Mobile App**
   - Download Pushover from your app store:
     - iOS App Store

  - Google Play Store
- Log in with your Pushover credentials
- Test the app by sending yourself a test message

3. **Note Your User Key**
   - Log into the Pushover web dashboard
   - Your User Key is displayed on the main page
   - Copy this key - you'll need it for Borgitory configuration

Step 2: Create Pushover Application

Register Application

1. **Create New Application**
   - In the Pushover dashboard, click "Create an Application/API Token"
   - Fill out the application details:

```
Name: Borgitory Backup Alerts
Type: Application
Description: Backup notifications from Borgitory
URL: http://localhost:8000 (or your Borgitory URL)
Icon: (optional - upload a custom icon)
```

2. **Get API Token**
   - After creating the application, note the API Token/Key
   - This token identifies your application to Pushover
   - Keep this token secure and don't share it publicly
3. **Test the Application**
   - Use the test feature in Pushover dashboard
   - Send a test message to verify everything works
   - Check that you receive the notification on your device

Step 3: Configure Borgitory

Add Pushover Configuration

1. **Navigate to Notifications**
   - Open Borgitory web interface
   - Go to Settings → Notifications
   - Click "Add Notification Configuration"
2. **Select Pushover Provider**
   - Choose "Pushover" from the provider dropdown
   - The Pushover-specific fields will appear
3. **Enter Configuration Details**

```
Name: Mobile Alerts
Provider: Pushover
User Key: [your-user-key-from-step-1]
API Token: [your-api-token-from-step-2]
Device: (optional - leave blank for all devices)
Priority: Normal
Sound: pushover (default)
```

Field Descriptions:

- **Name**: Friendly name for this notification config
- **User Key**: Your Pushover user key
- **API Token**: Your application's API token
- **Device**: Specific device name (optional - sends to all if blank)
- **Priority**: Notification priority (-2 to 2)
- **Sound**: Notification sound (see Pushover documentation)

4. **Test Configuration**
   - Click "Test Notification" button
   - You should receive a test message on your device
   - If the test fails, verify your User Key and API Token
5. **Save Configuration**
   - Click "Save" to store the notification configuration
   - The configuration appears in your notifications list

Step 4: Configure Notification Triggers

Global Notification Settings

1. **Set Default Preferences**
   - Go to Settings → Notifications → Global Settings
   - Configure default notification behavior:

```
Success Notifications: Enabled
Failure Notifications: Enabled
Warning Notifications: Enabled
Quiet Hours: 22:00 - 07:00 (optional)
Emergency Escalation: Disabled
```

Per-Schedule Configuration

1. **Edit Backup Schedules**
   - Go to Schedules
   - Click "Edit" on an existing schedule or create a new one
2. **Configure Notification Settings**
   - In the schedule form, find the "Notifications" section

- Enable notifications for this schedule
- Select your Pushover configuration
- Choose notification triggers:

```
✓ Notify on Success
✓ Notify on Failure
✓ Notify on Warning
 Notify on Start (optional)
```

3. **Custom Message Templates** (Optional)
   - Customize notification messages for this schedule
   - Use variables like `{schedule_name}`, `{status}`, `{duration}`
   - Example templates:

```
Success: " {schedule_name} completed in {duration}"
Failure: " {schedule_name} failed: {error_message}"
Warning: " {schedule_name} completed with warnings"
```

Step 5: Advanced Configuration

Priority Levels

Pushover supports different priority levels:

- **-2 (Lowest)**: No notification sound or vibration
- **-1 (Low)**: Quiet notification
- **0 (Normal)**: Default priority with sound
- **1 (High)**: High-priority sound and vibration
- **2 (Emergency)**: Repeated alerts until acknowledged

Configuration Example:

```
Success Notifications: Priority 0 (Normal)
Warning Notifications: Priority 1 (High)
Failure Notifications: Priority 2 (Emergency)
```

Device-Specific Notifications

To send notifications to specific devices:

1. **Find Device Names**
   - In Pushover app, go to Settings
   - Note the device name shown
   - Common names: “iPhone”, “Android”, “Desktop”
2. **Configure Device Targeting**
   - In Borgitory notification config, enter device name
   - Leave blank to send to all devices
   - Use comma-separated list for multiple devices: `iPhone,Desktop`

Custom Sounds

Pushover supports custom notification sounds:

Built-in Sounds: * `pushover` (default) * `bike`, `bugle`, `cashregister` * `classical`, `cosmic`, `falling` * `gamelan`, `incoming`, `intermission` * `magic`, `mechanical`, `pianobar` * `siren`, `spacealarm`, `tugboat` * `alien`, `climb`, `persistent` * `echo`, `updown`, `none` (silent)

Configuration:

```
Success Sound: pushover
Warning Sound: intermission
Failure Sound: siren
```

Quiet Hours

Configure quiet hours to avoid notifications during sleep:

```
Quiet Hours Start: 22:00
Quiet Hours End: 07:00
Emergency Override: Enabled (failures still notify)
```

Step 6: Testing and Validation

Test Notification Flow

1. **Manual Test**
   - Use the "Test Notification" button in configuration
   - Verify you receive the test message
   - Check that sound and priority work as expected
2. **Backup Test**
   - Run a manual backup job
   - Verify you receive success notification
   - Check notification content and timing
3. **Failure Test**
   - Create a backup with invalid source path
   - Verify you receive failure notification
   - Confirm emergency priority works (if configured)
4. **Schedule Test**
   - Create a test schedule that runs frequently
   - Wait for scheduled execution
   - Verify notifications arrive as expected

Troubleshooting Common Issues

No Notifications Received

Check Pushover Configuration:

```
# Test Pushover API directly
curl -s \
  --form-string "token=YOUR_API_TOKEN" \
  --form-string "user=YOUR_USER_KEY" \
  --form-string "message=Test from command line" \
  https://api.pushover.net/1/messages.json
```

Common Issues:

- **Invalid User Key**: Verify key from Pushover dashboard
- **Invalid API Token**: Verify token from application settings
- **Network Issues**: Check internet connectivity
- **Rate Limiting**: Pushover has API limits (7,500 messages/month free)

Notifications Not Working for Schedules

- **Schedule Not Enabled**: Verify schedule is active
- **Notification Config Not Selected**: Check schedule notification settings
- **Global Settings Override**: Review global notification preferences
- **Job Failures**: Check job logs for execution issues

Wrong Priority or Sound

- **Configuration Mismatch**: Verify priority/sound settings in Borgitory
- **Pushover App Settings**: Check notification settings in mobile app
- **Device-Specific Settings**: Some devices override notification settings

Delayed Notifications

- **Network Latency**: Pushover typically delivers within seconds
- **Device Sleep**: Some devices delay notifications when sleeping
- **App Background**: Ensure Pushover app isn't restricted in background

Step 7: Monitoring and Maintenance

Monitor Notification Health

1. **Check Delivery Status**
   - Pushover dashboard shows delivery statistics
   - Monitor successful vs. failed deliveries
   - Review API usage against limits
2. **Review Notification Logs**

- Borgitory logs notification attempts
- Check for API errors or timeouts
- Monitor notification frequency

3. **Update Configuration**
   - Periodically review notification settings
   - Adjust priorities based on experience
   - Update device names if changed

Quota Management

Free Tier Limits: * 7,500 messages per month * 10,000 API calls per month

Usage Optimization: * Use appropriate priorities to reduce noise * Configure quiet hours to batch notifications * Consider grouping related notifications * Monitor usage in Pushover dashboard

Upgrade Options: * Pushover Pro: $5 one-time fee for additional features * Higher quotas available for heavy users

Best Practices

Notification Strategy

- **Success Notifications**: Enable for critical schedules only
- **Failure Notifications**: Always enable for immediate attention
- **Warning Notifications**: Enable for troubleshooting
- **Quiet Hours**: Configure to avoid sleep disruption
- **Emergency Priority**: Reserve for critical failures only

Message Content

- **Be Specific**: Include schedule name, duration, error details
- **Use Emojis**: Visual indicators help quickly identify status
- **Keep Concise**: Mobile notifications have limited space
- **Include Context**: Timestamp, repository name, archive count

Security Considerations

- **Secure API Tokens**: Don't commit tokens to version control
- **Rotate Keys**: Periodically rotate API tokens
- **Limit Scope**: Use device targeting to limit notification scope
- **Monitor Usage**: Watch for unexpected API usage patterns

Next Steps

- Set up *How to Set Up Automated Backup Workflows* with notifications
- Configure *Monitoring Backup Health* for comprehensive monitoring
- Explore *How to Set Up AWS S3 Cloud Sync* for cloud backup with notifications

- Review *Troubleshooting Guide* for notification-related issues

With Pushover notifications configured, you'll receive immediate alerts about your backup operations, ensuring you stay informed about the health of your backup infrastructure even when away from your computer.

5.4.14 Monitoring Backup Health

This guide covers comprehensive monitoring strategies to ensure your backup infrastructure remains healthy and reliable.

Coming Soon

This guide will cover:

- Setting up health monitoring dashboards
- Key metrics to track
- Automated health checks
- Alert configuration
- Performance monitoring

For now, please refer to the *How to Set Up Automated Backup Workflows* guide for basic monitoring examples.

5.4.15 Performance Optimization Guide

This guide covers techniques to optimize Borgitory performance for large-scale backup operations.

Coming Soon

This guide will cover:

- System resource optimization
- Network performance tuning
- Storage optimization strategies
- Backup scheduling optimization
- Troubleshooting performance issues

For now, please refer to the *Docker Deployment Guide* guide for basic performance considerations.

5.4.16 Windows WSL Setup Guide

This guide provides step-by-step instructions for setting up Borgitory on a fresh Windows installation using WSL2 (Windows Subsystem for Linux).

Prerequisites

- Windows 10 version 2004 and higher (Build 19041 and higher) or Windows 11
- Administrator access to your Windows machine
- Internet connection

Step 1: Enable WSL2

Refer to the Microsoft documentation for more details: https://learn.microsoft.com/en-us/windows/wsl/install

```
# Enable WSL feature
wsl --install
```

Step 2: Install Ubuntu Linux Distribution

Install Ubuntu

```
# List available distributions
wsl --list --online

# Install Ubuntu (replace with your preferred version)
wsl --install -d Ubuntu-22.04
```

Initial Ubuntu Setup

1. Launch Ubuntu from Start Menu
2. Create a new user account when prompted
3. Set a password for your user account

Step 3: Update Ubuntu and Install Dependencies

Update Package Lists

```
sudo apt update && sudo apt upgrade -y
```

Install Python 3.11+

```
# Install Python 3.11 and pip
sudo apt install software-properties-common -y
sudo add-apt-repository ppa:deadsnakes/ppa -y
sudo apt update
sudo apt install python3.11 python3.11-pip python3.11-venv -y

# Set Python 3.11 as default (optional)
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.11 1
```

Install BorgBackup

```
# Install BorgBackup
sudo apt install borgbackup -y

# Verify installation
borg --version
```

Install Rclone (Optional - for cloud sync)

```
# Download and install rclone
curl https://rclone.org/install.sh | sudo bash

# Verify installation
rclone version
```

Install FUSE Support (Optional - for archive browsing)

```
# Install FUSE3 development libraries
sudo apt install fuse3 libfuse3-dev -y

# Install Python FUSE bindings
python3.11 -m pip install pyfuse3
```

Step 4: Install Borgitory

Install Borgitory via pip

```
# Install Borgitory
python3.11 -m pip install borgitory

# Verify installation
borgitory --help
```

Create Directory Structure

```
# Create directories for Borgitory data
mkdir -p ~/borgitory/{data,repos,sources}

# Set appropriate permissions
chmod 755 ~/borgitory/{data,repos,sources}
```

Step 5: Configure Windows-WSL Integration

Access Windows Files from WSL

Windows drives are automatically mounted under `/mnt/`:

- C: drive → `/mnt/c/`
- D: drive → `/mnt/d/`

Create Symbolic Links for Easy Access

```
# Link to common Windows directories
ln -s /mnt/c/Users/$USER/Documents ~/Documents-Windows
ln -s /mnt/c/Users/$USER/Desktop ~/Desktop-Windows

# Example: Link backup sources
ln -s /mnt/c/Users/$USER/Documents ~/borgitory/sources/documents
```

Step 6: Start Borgitory

Run Borgitory Server

```
# Start Borgitory with default settings
borgitory serve

# Or with custom host/port
borgitory serve --host 0.0.0.0 --port 8000
```

Access Web Interface

1. Open your web browser
2. Navigate to `http://localhost:8000`
3. Create your first admin account on initial setup

Step 7: Optional Configurations

Create Windows Shortcut

Create a batch file to easily start Borgitory:

1. Create `start-borgitory.bat` on your Windows desktop:

```
@echo off
wsl -d Ubuntu-22.04 -e bash -c "cd ~ && borgitory serve --host 0.0.0.0 --port 8000"
```

2. Double-click the batch file to start Borgitory

Auto-start WSL on Windows Boot (Optional)

1. Open Task Scheduler
2. Create Basic Task
3. Set trigger to "When the computer starts"
4. Set action to start program: `wsl.exe`
5. Add arguments: `-d Ubuntu-22.04`

Troubleshooting

Common Issues

1. **WSL2 not starting**: Ensure virtualization is enabled in BIOS
2. **Permission denied errors**: Check file permissions and ownership
3. **Network issues**: WSL2 uses NAT networking by default
4. **FUSE mount failures**: Ensure FUSE is properly installed and user has permissions

Useful Commands

```
# Check WSL version
wsl --list --verbose

# Restart WSL
wsl --shutdown
wsl

# Check if services are running
systemctl status fuse3  # if using systemd
```

Getting Help

- WSL Documentation
- BorgBackup Documentation
- Rclone Documentation
- *Troubleshooting Guide* - Borgitory troubleshooting guide

Security Considerations

1. **File Permissions**: Be careful with file permissions between Windows and WSL
2. **Network Access**: Consider firewall rules if accessing from other machines
3. **Backup Encryption**: Always use encrypted BorgBackup repositories
4. **WSL Security**: Keep WSL and Ubuntu updated regularly

Note

This setup allows you to run Borgitory on Windows by leveraging WSL2 for Linux compatibility. All backup operations will run within the WSL2 environment while providing a web interface accessible from Windows.

Tip

For production use, consider the *Docker Deployment Guide* which provides better isolation and easier management, even on Windows systems with WSL2.

5.4.17 Docker Deployment Guide

This guide covers production deployment of Borgitory using Docker and Docker Compose, including best practices for security, performance, and maintenance.

Overview

Borgitory's Docker deployment provides:

- Isolated environment with all dependencies
- FUSE support for archive browsing
- Persistent data storage

- Easy updates and rollbacks
- Scalable configuration

Production Requirements

System Requirements

Minimum Requirements:

- 2 CPU cores
- 4 GB RAM
- 20 GB disk space (for application)
- Additional storage for repositories and backups

Recommended for Production:

- 4+ CPU cores
- 8+ GB RAM
- SSD storage for application data
- Separate storage volumes for repositories

Required Docker Capabilities:

- `SYS_ADMIN` capability for FUSE mounting
- `/dev/fuse` device access for archive browsing
- Volume mounting for persistent storage

Network Requirements

- **Port 8000**: Web interface access
- **Outbound HTTPS (443)**: Cloud sync operations
- **Outbound SSH (22)**: Remote repository access (if used)
- **DNS Resolution**: For cloud provider APIs

Production Docker Compose

Complete Production Configuration

Create `docker-compose.prod.yml`:

```yaml
version: '3.8'

services:
  borgitory:
    image: mlapaglia/borgitory:latest
    container_name: borgitory-prod
    restart: unless-stopped

    # Network configuration
    ports:
      - "8000:8000"
```

(continues on next page)

(continued from previous page)

```
    # Security and capabilities
    cap_add:
      - SYS_ADMIN
    devices:
      - /dev/fuse
    security_opt:
      - apparmor:unconfined

    # Environment variables
    environment:
      - BORGITORY_HOST=0.0.0.0
      - BORGITORY_PORT=8000
      - BORGITORY_DATA_DIR=/app/data
      - BORGITORY_DEBUG=false
      - TZ=America/New_York

    # Volume mounts
    volumes:
      # Application data (persistent)
      - ./data:/app/data

      # Backup sources (read-only)
      - /srv/backup-sources:/mnt/backup/sources:ro
      - /home:/mnt/backup/home:ro
      - /etc:/mnt/backup/etc:ro

      # Repository storage (read-write)
      - /srv/borg-repos:/mnt/repos
      - /mnt/external-drive/repos:/mnt/external-repos

      # Optional: SSH keys for remote repositories
      - ~/.ssh:/root/.ssh:ro

    # Resource limits
    deploy:
      resources:
        limits:
          memory: 2G
          cpus: '2.0'
        reservations:
          memory: 1G
          cpus: '1.0'

    # Health check
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 60s
```

(continues on next page)

(continued from previous page)

```
    # Logging configuration
    logging:
      driver: "json-file"
      options:
        max-size: "100m"
        max-file: "5"

# Optional: Reverse proxy with SSL
networks:
  default:
    name: borgitory-network
```

Advanced Configuration Options

With Traefik Reverse Proxy:

```
version: '3.8'

services:
  borgitory:
    image: mlapaglia/borgitory:latest
    container_name: borgitory-prod
    restart: unless-stopped

    cap_add:
      - SYS_ADMIN
    devices:
      - /dev/fuse

    volumes:
      - ./data:/app/data
      - /srv/backup-sources:/mnt/backup/sources:ro
      - /srv/borg-repos:/mnt/repos

    networks:
      - traefik

    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.borgitory.rule=Host(`backups.yourdomain.com`)"
      - "traefik.http.routers.borgitory.tls=true"
      - "traefik.http.routers.borgitory.tls.certresolver=letsencrypt"
      - "traefik.http.services.borgitory.loadbalancer.server.port=8000"

networks:
  traefik:
    external: true
```

Storage Configuration

Volume Mapping Strategy

Application Data Volume:

```
volumes:
  - ./data:/app/data
```

Contains:

- SQLite database
- Configuration files
- Encryption keys
- Job history

Backup Source Volumes:

```
volumes:
  - /srv/backup-sources:/mnt/backup/sources:ro
  - /home:/mnt/backup/home:ro
  - /var/log:/mnt/backup/logs:ro
```

- Mount as read-only (`:ro`) for safety
- Map all directories you want to backup
- Use descriptive paths under `/mnt/`

Repository Storage Volumes:

```
volumes:
  - /srv/borg-repos:/mnt/repos
  - /mnt/nas/backups:/mnt/nas-repos
  - /mnt/external-drive:/mnt/external-repos
```

- Read-write access required
- Can be local or network storage
- Consider redundancy and backup strategies

Directory Structure Example

```
/srv/
├── borgitory/
│   ├── docker-compose.prod.yml
│   ├── data/                    # Application data volume
│   │   ├── borgitory.db
│   │   └── secret_key
│   └── backups/                 # Backup logs and exports
├── backup-sources/              # Source data to backup
│   ├── documents/
│   ├── photos/
│   └── projects/
└── borg-repos/                  # Borg repository storage
```

(continues on next page)

(continued from previous page)

```
    ├── documents-repo/
    ├── photos-repo/
    └── system-repo/
```

Security Configuration

User and Permissions

Run as Non-Root User:

```yaml
services:
  borgitory:
    user: "1000:1000"  # Replace with your user ID
    # ... other configuration
```

File Permissions:

```
# Set ownership of data directory
sudo chown -R 1000:1000 ./data

# Set secure permissions
chmod 700 ./data
chmod 600 ./data/secret_key
```

Network Security

Firewall Configuration:

```
# Allow only necessary ports
sudo ufw allow 8000/tcp  # Borgitory web interface
sudo ufw deny 8000/tcp from any to any  # Deny external access

# Or with specific source
sudo ufw allow from 192.168.1.0/24 to any port 8000
```

Reverse Proxy with SSL:

Use Traefik, Nginx, or Apache for SSL termination:

```
server {
    listen 443 ssl;
    server_name backups.yourdomain.com;

    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    location / {
        proxy_pass http://localhost:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Backup and Recovery

Application Data Backup

Database Backup:

```
#!/bin/bash
# backup-borgitory.sh

DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_DIR="/srv/borgitory-backups"
DATA_DIR="/srv/borgitory/data"

# Create backup directory
mkdir -p "$BACKUP_DIR"

# Stop container for consistent backup
docker-compose -f docker-compose.prod.yml stop borgitory

# Create backup archive
tar -czf "$BACKUP_DIR/borgitory-backup-$DATE.tar.gz" -C "$DATA_DIR" .

# Start container
docker-compose -f docker-compose.prod.yml start borgitory

# Clean old backups (keep 30 days)
find "$BACKUP_DIR" -name "borgitory-backup-*.tar.gz" -mtime +30 -delete
```

Automated Backup with Cron:

```
# Add to crontab
0 2 * * * /srv/borgitory/backup-borgitory.sh
```

Recovery Procedures

Application Recovery:

```
# Stop container
docker-compose -f docker-compose.prod.yml stop borgitory

# Restore from backup
cd /srv/borgitory
tar -xzf /srv/borgitory-backups/borgitory-backup-YYYYMMDD_HHMMSS.tar.gz -C data/

# Fix permissions
chown -R 1000:1000 data/

# Start container
docker-compose -f docker-compose.prod.yml start borgitory
```

Disaster Recovery:

```
# Complete system restore
# 1. Install Docker and Docker Compose
# 2. Restore application data
# 3. Restore repository data (from offsite backups)
# 4. Start services

docker-compose -f docker-compose.prod.yml up -d
```

Monitoring and Maintenance

Health Monitoring

Docker Health Checks:

```
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 60s
```

External Monitoring:

```
#!/bin/bash
# monitor-borgitory.sh

URL="http://localhost:8000/health"

if ! curl -f -s "$URL" > /dev/null; then
    echo "Borgitory health check failed" | mail -s "Borgitory Alert" admin@example.com
    # Optionally restart service
    docker-compose -f docker-compose.prod.yml restart borgitory
fi
```

Log Management

Log Rotation:

```
logging:
  driver: "json-file"
  options:
    max-size: "100m"
    max-file: "5"
```

Centralized Logging:

```
logging:
  driver: "syslog"
  options:
    syslog-address: "tcp://logserver:514"
    tag: "borgitory"
```

Updates and Maintenance

Update Procedure:

```
#!/bin/bash
# update-borgitory.sh

cd /srv/borgitory

# Backup current data
./backup-borgitory.sh

# Pull latest image
docker-compose -f docker-compose.prod.yml pull

# Recreate containers
docker-compose -f docker-compose.prod.yml up -d

# Check health
sleep 30
docker-compose -f docker-compose.prod.yml ps
curl -f http://localhost:8000/health
```

Automated Updates:

```
# Add to crontab for weekly updates
0 3 * * 0 /srv/borgitory/update-borgitory.sh
```

Performance Optimization

Resource Allocation

CPU Optimization:

```
deploy:
  resources:
    limits:
      cpus: '4.0'       # Adjust based on backup workload
    reservations:
      cpus: '2.0'
```

Memory Optimization:

```
deploy:
  resources:
    limits:
      memory: 4G        # Increase for large repositories
    reservations:
      memory: 2G
```

Storage Performance

SSD for Application Data:

- Use SSD for `/app/data` volume
- Improves database performance
- Faster job processing

Network Storage Considerations:

- Use local storage for frequently accessed repositories
- Network storage acceptable for archive storage
- Consider NFS vs. CIFS performance

Troubleshooting

Common Issues

FUSE Mount Failures:

```
# Check FUSE availability
ls -la /dev/fuse

# Verify container capabilities
docker inspect borgitory-prod | grep -i cap

# Check kernel modules
lsmod | grep fuse
```

Permission Errors:

```
# Fix data directory permissions
sudo chown -R 1000:1000 ./data

# Check volume mount permissions
ls -la /srv/backup-sources
ls -la /srv/borg-repos
```

Container Won't Start:

```
# Check logs
docker-compose -f docker-compose.prod.yml logs borgitory

# Check system resources
df -h
free -h

# Verify configuration
docker-compose -f docker-compose.prod.yml config
```

Performance Issues

Slow Backups:

- Check disk I/O: `iostat -x 1`
- Monitor network: `iftop`
- Review compression settings
- Consider parallel operations

High Memory Usage:

- Increase memory limits
- Monitor with `docker stats`
- Check for memory leaks in logs

Database Performance:

- Vacuum SQLite database periodically
- Monitor database size
- Consider moving to PostgreSQL for large deployments

Next Steps

- Set up *Monitoring Backup Health* for comprehensive monitoring
- Configure *How to Set Up Pushover Notifications* for alerts
- Implement *How to Set Up Automated Backup Workflows* for production schedules
- Review *Performance Optimization Guide* for tuning
- Plan *Multi-Cloud Sync Setup Guide* for redundancy

5.4.18 Custom Compression Settings Guide

This guide explains how to configure and optimize compression settings for different types of data and use cases.

Coming Soon

This guide will cover:

- Understanding compression algorithms (lz4, zlib, lzma, zstd)
- Choosing compression based on data types
- Performance vs. compression ratio trade-offs
- Custom compression configurations
- Testing and benchmarking compression settings

For now, please refer to the *Usage Guide* guide for basic compression options.

5.4.19 Network Storage Integration Guide

This guide covers integrating Borgitory with various network storage solutions like NFS, SMB/CIFS, and iSCSI.

Coming Soon

This guide will cover:

- NFS mount configuration
- SMB/CIFS integration
- iSCSI storage setup
- Performance considerations for network storage
- Security best practices

For now, please refer to the *Docker Deployment Guide* guide for volume mounting strategies.

5.4.20 Package Manager for Pre/Post Hooks

The Borgitory package manager allows you to install Debian packages directly from the web interface to support your pre and post job hooks. This feature enables you to extend the functionality of your backup workflows by installing the tools your custom scripts need.

- *Overview*
- *Accessing the Package Manager*
- *Installing Packages*
  - *Searching for Packages*
  - *Selecting and Installing Packages*
- *Managing Installed Packages*
- *Package Persistence*
- *Integration with Pre/Post Hooks*
  - *Accessing Hook Configuration*
- *Common Use Cases and Examples*
  - *Database Dumps with SQLite3*
  - *PostgreSQL Database Dumps*
  - *Web Content Downloads with curl*
  - *Log Processing with jq*
  - *System Health Checks*
  - *Notification Integrations*
- *Advanced Hook Features*
  - *Critical Hooks*
  - *Run on Job Failure*
  - *Environment Variables in Hooks*
- *Best Practices*
  - *Package Selection*

- *Hook Development*
- *Security Considerations*
- *Performance Tips*

- *Troubleshooting*
  - *Common Issues*
  - *Viewing Logs*
  - *Getting Help*

Overview

The package manager provides a secure way to install additional software packages into the Borgitory Docker container. These packages persist across container restarts and are automatically reinstalled if needed. The primary use case is to support pre-hook and post-hook scripts that require additional command-line tools.

> **Note**
>
> The package manager only works when Borgitory is running in a Docker container with appropriate permissions. Packages are installed using the standard `apt` package manager.

Accessing the Package Manager

1. Navigate to the main Borgitory interface
2. Click on the **"Package Manager"** tab in the navigation menu
3. The package manager interface will load, showing two main sections: - **Install Packages** - Search and install new packages - **Installed Packages** - View and manage currently installed packages

Installing Packages

Searching for Packages

1. In the **Install Packages** section, use the search box to find packages
2. Type at least 2 characters to start searching
3. An autocomplete dropdown will appear showing matching packages
4. Each result shows: - Package name and version - Brief description - Section category - Installation status (if already installed)

Selecting and Installing Packages

1. Click on a package from the search results to select it
2. Selected packages appear in the **Selected Packages** area
3. You can select multiple packages before installing
4. Remove packages from selection by clicking the × button
5. Click **"Install Selected Packages"** to begin installation
6. Installation progress and results will be displayed

Managing Installed Packages

The **Installed Packages** section shows all packages currently installed in the container. Packages are categorized as:

- **System Packages** - Pre-installed packages (cannot be removed)
- **User-Installed Packages** - Packages you installed (can be removed)

User-installed packages have a **Remove** button that allows you to uninstall them if they're no longer needed.

Package Persistence

Packages installed through the package manager are automatically tracked and will be reinstalled if:

- The container is restarted
- The container image is updated
- The package is accidentally removed

This ensures your pre/post hook scripts continue to work reliably across deployments.

Integration with Pre/Post Hooks

The package manager is specifically designed to support pre and post job hooks. When you create or edit a schedule, you can configure hooks that run before and after backup operations.

Accessing Hook Configuration

1. Go to **Schedules** and create or edit a schedule
2. In the schedule form, look for the **"Pre/Post Job Hooks"** section
3. Click **"Configure Hooks"** to open the hooks modal
4. Add pre-hooks (run before backup) and post-hooks (run after backup)

Common Use Cases and Examples

Database Dumps with SQLite3

Scenario: Backup SQLite database files by creating dumps before the backup runs.

Required Package: `sqlite3`

Installation: 1. Search for "sqlite3" in the package manager 2. Select and install the `sqlite3` package

Pre-hook Example:

```
#!/bin/bash
# Pre-hook: Create SQLite database dump

# Define paths
DB_PATH="/app/data/myapp.db"
DUMP_PATH="/app/data/myapp_dump.sql"

# Create dump if database exists
if [ -f "$DB_PATH" ]; then
    echo "Creating SQLite dump..."
    sqlite3 "$DB_PATH" .dump > "$DUMP_PATH"
    echo "SQLite dump created: $DUMP_PATH"
```

(continues on next page)

(continued from previous page)

```
else
    echo "Database not found: $DB_PATH"
    exit 1
fi
```

Benefits: - Ensures consistent database state during backup - Creates human-readable SQL dumps - Can be restored on any system with SQLite

PostgreSQL Database Dumps

Scenario: Create PostgreSQL database dumps before backup.

Required Packages: `postgresql-client`

Installation: 1. Search for "postgresql-client" 2. Install the `postgresql-client` package

Pre-hook Example:

```
#!/bin/bash
# Pre-hook: Create PostgreSQL database dump

# Database connection details
DB_HOST="localhost"
DB_NAME="myapp"
DB_USER="backup_user"
DUMP_PATH="/app/data/postgres_dump.sql"

# Create dump
echo "Creating PostgreSQL dump..."
pg_dump -h "$DB_HOST" -U "$DB_USER" -d "$DB_NAME" > "$DUMP_PATH"

if [ $? -eq 0 ]; then
    echo "PostgreSQL dump created successfully"
else
    echo "PostgreSQL dump failed"
    exit 1
fi
```

Post-hook Example:

```
#!/bin/bash
# Post-hook: Clean up dump file after backup

DUMP_PATH="/app/data/postgres_dump.sql"

if [ -f "$DUMP_PATH" ]; then
    rm "$DUMP_PATH"
    echo "Cleaned up PostgreSQL dump file"
fi
```

Web Content Downloads with curl

Scenario: Download important web content or API data before backup.

Required Package: `curl`

Pre-hook Example:

```
#!/bin/bash
# Pre-hook: Download important web content

BACKUP_DIR="/app/data/web_backups"
mkdir -p "$BACKUP_DIR"

# Download website content
echo "Downloading website content..."
curl -o "$BACKUP_DIR/website.html" "https://mywebsite.com/important-page"

# Download API data
echo "Downloading API data..."
curl -H "Authorization: Bearer $API_TOKEN" \
     -o "$BACKUP_DIR/api_data.json" \
     "https://api.myservice.com/data"

echo "Web content downloads completed"
```

Log Processing with jq

Scenario: Process and filter JSON logs before backup.

Required Package: `jq`

Pre-hook Example:

```
#!/bin/bash
# Pre-hook: Process JSON logs

LOG_DIR="/app/logs"
PROCESSED_DIR="/app/data/processed_logs"
mkdir -p "$PROCESSED_DIR"

# Process JSON logs to extract important events
echo "Processing JSON logs..."
for log_file in "$LOG_DIR"/*.json; do
    if [ -f "$log_file" ]; then
        filename=$(basename "$log_file")
        # Extract error events only
        jq '.[] | select(.level == "error")' "$log_file" > "$PROCESSED_DIR/errors_
↪$filename"
        # Extract user activity
        jq '.[] | select(.type == "user_activity")' "$log_file" > "$PROCESSED_DIR/
↪activity_$filename"
    fi
done

echo "Log processing completed"
```

System Health Checks

Scenario: Perform system health checks before backup and fail if system is unhealthy.

Required Packages: `curl`, `jq`

Pre-hook Example (with Critical flag enabled):

```
#!/bin/bash
# Pre-hook: System health check (CRITICAL)
# This hook should be marked as "Critical" to stop backup if health check fails

echo "Performing system health check..."

# Check disk space
DISK_USAGE=$(df /app | tail -1 | awk '{print $5}' | sed 's/%//')
if [ "$DISK_USAGE" -gt 90 ]; then
    echo "ERROR: Disk usage is ${DISK_USAGE}% - too high for safe backup"
    exit 1
fi

# Check application health endpoint
HTTP_STATUS=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/health)
if [ "$HTTP_STATUS" != "200" ]; then
    echo "ERROR: Application health check failed (HTTP $HTTP_STATUS)"
    exit 1
fi

# Check database connectivity
if ! pg_isready -h localhost -p 5432; then
    echo "ERROR: Database is not ready"
    exit 1
fi

echo "System health check passed"
```

Notification Integrations

Scenario: Send custom notifications about backup status.

Required Package: `curl`

Post-hook Example (with "Run on job failure" enabled):

```
#!/bin/bash
# Post-hook: Send Slack notification
# This hook should have "Run on job failure" enabled to notify about failures

SLACK_WEBHOOK_URL="https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK"

# Check if backup was successful by checking environment variables
# (These are provided by Borgitory to hook scripts)
if [ "$BORGITORY_JOB_STATUS" = "success" ]; then
    MESSAGE=" Backup completed successfully for repository: $BORGITORY_REPOSITORY_ID"
    COLOR="good"
```

(continues on next page)

(continued from previous page)

```
else
    MESSAGE=" Backup failed for repository: $BORGITORY_REPOSITORY_ID"
    COLOR="danger"
fi

# Send Slack notification
curl -X POST -H 'Content-type: application/json' \
     --data "{\"text\":\"$MESSAGE\", \"color\":\"$COLOR\"}" \
     "$SLACK_WEBHOOK_URL"

echo "Notification sent to Slack"
```

Advanced Hook Features

Critical Hooks

When configuring hooks, you can mark them as **"Critical"**. If a critical hook fails:

- The entire backup job stops immediately
- Remaining hooks and backup tasks are skipped
- The job is marked as failed
- Notifications still run (if configured to run on failure)

This is useful for: - Health checks that must pass before backup - Database dumps that are essential for data consistency - Security checks that should prevent backup if failed

Run on Job Failure

Post-hooks can be configured to **"Run on job failure"**. These hooks will execute even if:

- A critical pre-hook failed
- The backup process failed
- Any other part of the job failed

This is useful for: - Cleanup operations that should always run - Failure notifications - Logging and monitoring - Releasing locks or resources

Environment Variables in Hooks

Borgitory provides several environment variables to your hook scripts:

- `BORGITORY_REPOSITORY_ID` - The ID of the repository being backed up
- `BORGITORY_JOB_TYPE` - The type of job (e.g., "backup", "prune")
- `BORGITORY_TASK_INDEX` - The current task number in the job
- `BORGITORY_JOB_STATUS` - Current job status (available in post-hooks)

Example using environment variables:

```
#!/bin/bash
# Hook script using Borgitory environment variables
```

(continues on next page)

(continued from previous page)

```
echo "Processing backup for repository: $BORGITORY_REPOSITORY_ID"
echo "Job type: $BORGITORY_JOB_TYPE"
echo "Current task: $BORGITORY_TASK_INDEX"

# Create repository-specific backup directory
BACKUP_DIR="/app/data/repo_$BORGITORY_REPOSITORY_ID"
mkdir -p "$BACKUP_DIR"

# Perform repository-specific operations...
```

Best Practices

Package Selection

- **Install only what you need**: Each package increases container size and potential security surface
- **Use specific packages**: Choose `postgresql-client` over `postgresql` if you only need client tools
- **Check dependencies**: Some packages install many dependencies automatically
- **Regular cleanup**: Remove packages that are no longer needed

Hook Development

- **Test thoroughly**: Test hooks in a development environment before production
- **Handle errors gracefully**: Use proper exit codes and error messages
- **Log operations**: Include echo statements to track hook execution
- **Use absolute paths**: Don't rely on relative paths or current directory
- **Check prerequisites**: Verify required files/services exist before processing

Security Considerations

- **Validate inputs**: Don't trust external data sources
- **Use secure connections**: Always use HTTPS/TLS for external communications
- **Protect credentials**: Use environment variables or secure files for API keys
- **Limit permissions**: Run with minimum required privileges
- **Regular updates**: Keep installed packages updated

Performance Tips

- **Minimize hook runtime**: Long-running hooks delay backup completion
- **Parallel processing**: Use background jobs for independent operations
- **Efficient tools**: Choose lightweight tools when possible
- **Cache data**: Avoid re-downloading the same data repeatedly
- **Clean up**: Remove temporary files to save space

Troubleshooting

Common Issues

Package not found

- Check spelling and package name
- Update package lists (this happens automatically)
- Verify the package exists in Debian repositories

Installation fails

- Check container permissions
- Verify internet connectivity
- Look for dependency conflicts in logs

Hook script fails

- Check script permissions (should be executable)
- Verify required packages are installed
- Check script syntax and logic
- Review hook execution logs

Packages disappear after restart

- This shouldn't happen with the persistence system
- Check container volume mounts
- Verify database connectivity

Viewing Logs

Hook execution is logged in the job details. To view hook logs:

1. Go to **Jobs** in the navigation
2. Find your backup job in the list
3. Click to expand job details
4. Look for hook tasks in the task list
5. Click on individual hook tasks to see their output

Getting Help

If you encounter issues with the package manager or hooks:

1. Check the application logs for error messages
2. Verify your hook scripts work outside of Borgitory
3. Test package installation manually in the container
4. Review this documentation for examples and best practices
5. Check the Borgitory GitHub repository for known issues

Remember that the package manager and hooks system is designed to be flexible and powerful. Start with simple examples and gradually build more complex workflows as you become comfortable with the system.

5.4.21 Quick Reference

Common Tasks

- *Backup Patterns Guide* - Configure include/exclude patterns for backups
- *Windows WSL Setup Guide* - Complete Windows setup using WSL2
- *Docker Deployment Guide* - Production Docker deployment
- *How to Set Up Multiple Repositories* - Managing multiple backup destinations
- *How to Set Up Automated Backup Workflows* - Setting up comprehensive backup automation
- *How to Set Up AWS S3 Cloud Sync* - Configuring Amazon S3 for cloud backups
- *How to Set Up Pushover Notifications* - Getting alerts on your mobile device
- *Package Manager for Pre/Post Hooks* - Installing tools for pre/post hooks

Troubleshooting

- *Troubleshooting Guide* - Common issues and solutions
- *Repository Maintenance Guide* - Keeping repositories healthy
- *Performance Optimization Guide* - Improving backup performance

5.4.22 Need Help?

- GitHub Issues - Report bugs or request features
- GitHub Discussions - Community support
- *Troubleshooting Guide* - Detailed troubleshooting guide

5.5 Troubleshooting Guide

This guide covers common issues, error messages, and solutions when using Borgitory.

5.5.1 Quick Diagnostics

System Health Check

Before troubleshooting specific issues, verify your system meets the basic requirements:

Check BorgBackup Installation:

```
# Verify Borg is installed and accessible
borg --version

# Check PATH includes Borg location
which borg

# Test basic Borg functionality
borg help
```

Check Rclone Installation (for cloud sync):

```
# Verify Rclone is installed
rclone --version

# List configured remotes
rclone listremotes

# Test basic functionality
rclone help
```

Check System Resources:

```
# Check disk space
df -h

# Check memory usage
free -h

# Check CPU usage
top
```

Docker-Specific Checks:

```
# Check container status
docker ps

# Check container logs
docker logs borgitory

# Check FUSE availability
ls -la /dev/fuse
```

5.5.2 Installation Issues

PyPI Installation Problems

Python Version Compatibility

```
Error: borgitory requires Python 3.11 or higher
```

Solution:

```
# Check Python version
python --version

# Install Python 3.11+ using pyenv (recommended)
curl https://pyenv.run | bash
pyenv install 3.11.0
pyenv global 3.11.0

# Or use system package manager
# Ubuntu/Debian:
sudo apt update
sudo apt install python3.11 python3.11-pip
```

(continues on next page)

(continued from previous page)

```
# Create virtual environment with correct Python
python3.11 -m venv .venv
source .venv/bin/activate
pip install borgitory
```

Missing System Dependencies

```
Error: Failed building wheel for pyfuse3
```

Solution:

```
# Ubuntu/Debian
sudo apt update
sudo apt install python3-dev libfuse3-dev pkg-config

# CentOS/RHEL/Fedora
sudo dnf install python3-devel fuse3-devel pkgconfig

# macOS
brew install macfuse pkg-config
```

Permission Errors During Installation

```
Error: Permission denied when installing borgitory
```

Solution:

```
# Use virtual environment (recommended)
python -m venv .venv
source .venv/bin/activate
pip install borgitory

# Or install for user only
pip install --user borgitory
```

Docker Installation Problems

FUSE Mount Failures

```
Error: Archive browsing disabled - FUSE not available
```

Solution:

```
# Ensure container runs with proper capabilities
docker run -d \
  --cap-add SYS_ADMIN \
  --device /dev/fuse \
  mlapaglia/borgitory:latest

# Check if FUSE is available on host
ls -la /dev/fuse
```

(continues on next page)

(continued from previous page)

```
# Install FUSE if missing
# Ubuntu/Debian:
sudo apt install fuse3

# Load FUSE kernel module if needed
sudo modprobe fuse
```

Volume Mount Issues

```
Error: No such file or directory when accessing backup sources
```

Solution:

```
# Verify source paths exist on host
ls -la /path/to/backup/sources

# Check volume mounts in Docker Compose
volumes:
  - /actual/host/path:/mnt/container/path:ro

# Ensure paths are under /mnt/ in container
- /home/user/documents:/mnt/backup/documents:ro
```

5.5.3 Repository Issues

Repository Connection Failures

Invalid Repository Path

```
Error: Repository not found or inaccessible
```

Diagnosis:

```
# Test repository access directly with Borg
borg info /path/to/repository

# Check if path exists and is accessible
ls -la /path/to/repository

# Verify permissions
stat /path/to/repository
```

Solutions:

- Verify the repository path is correct
- Ensure the path is accessible from within the container (use `/mnt/` prefix for Docker)
- Check file permissions allow read/write access
- For remote repositories, verify SSH keys and network connectivity

Incorrect Passphrase

```
Error: Invalid passphrase or corrupted repository
```

Diagnosis:

```
# Test passphrase directly with Borg
borg list /path/to/repository
# Enter passphrase when prompted
```

Solutions:

- Verify the passphrase is exactly correct (case-sensitive)
- Check for hidden characters or encoding issues
- Try accessing the repository with Borg CLI to confirm passphrase
- If passphrase is lost, repository cannot be recovered

Repository Corruption

```
Error: Repository integrity check failed
```

Diagnosis:

```
# Check repository integrity
borg check /path/to/repository

# Detailed check with repair option
borg check --repair /path/to/repository
```

Solutions:

- Run `borg check --repair` to attempt automatic repair
- If repair fails, restore from backup repository
- Check underlying storage for hardware issues
- Consider filesystem corruption on storage device

5.5.4 Backup Operation Failures

Backup Job Failures

Insufficient Disk Space

```
Error: No space left on device
```

Diagnosis:

```
# Check available space
df -h

# Check repository location specifically
df -h /path/to/repository
```

Solutions:

- Free up disk space on repository storage
- Move repository to location with more space
- Implement pruning policies to remove old archives

- Consider using compression to reduce space usage

Permission Denied Errors

```
Error: Permission denied accessing source files
```

Diagnosis:

```
# Check source path permissions
ls -la /path/to/source

# Test read access
cat /path/to/source/testfile
```

Solutions:

- Ensure Borgitory has read access to source directories
- For Docker, check volume mount permissions
- Run container with appropriate user ID
- Use `sudo` if necessary for system directories

Exclude Pattern Issues

```
Warning: Exclude pattern not matching any files
```

Diagnosis:

- Review exclude patterns for syntax errors
- Test patterns with find command
- Check if patterns are relative to source path

Solutions:

```
# Test exclude patterns
find /path/to/source -name "*.tmp"

# Common exclude patterns
*.tmp
*.log
.git/
__pycache__/
node_modules/
```

5.5.5 Cloud Sync Issues

Connection Failures

Invalid Credentials

```
Error: Access denied (403) - Invalid credentials
```

Diagnosis:

```
# Test credentials with rclone directly
rclone lsd remote:bucket

# Check credential format and validity
rclone config show remote
```

Solutions:

- Verify access keys are correct and not expired
- Check if credentials have necessary permissions
- For AWS S3, verify IAM policy allows required actions
- Test credentials outside Borgitory first

Network Connectivity Issues

```
Error: Connection timeout or network unreachable
```

Diagnosis:

```
# Test network connectivity
ping google.com

# Test HTTPS connectivity
curl -I https://s3.amazonaws.com

# Check DNS resolution
nslookup s3.amazonaws.com
```

Solutions:

- Verify internet connectivity
- Check firewall rules allow outbound HTTPS
- Configure proxy settings if behind corporate firewall
- Verify DNS resolution works correctly

Bucket/Container Access Issues

```
Error: Bucket does not exist or access denied
```

Solutions:

- Verify bucket/container name is correct
- Check if bucket exists in specified region
- Ensure credentials have access to the specific bucket
- Review bucket policies and access controls

5.5.6 Performance Issues

Slow Backup Operations

Disk I/O Bottlenecks

Diagnosis:

```
# Monitor disk I/O
iostat -x 1

# Check disk usage patterns
iotop
```

Solutions:

- Use SSD storage for repositories when possible
- Avoid running multiple intensive operations simultaneously
- Consider different compression algorithms (lz4 for speed, lzma for size)
- Spread repositories across multiple disks

Network Performance Issues

Diagnosis:

```
# Test network speed
speedtest-cli

# Monitor network usage
iftop

# Test cloud provider connectivity
rclone test speed remote:
```

Solutions:

- Choose cloud regions closer to your location
- Use appropriate compression for your network speed
- Consider bandwidth limiting during peak hours
- Implement incremental backup strategies

High Memory Usage

Large Repository Memory Consumption

Diagnosis:

```
# Monitor memory usage
free -h

# Check process memory usage
ps aux --sort=-%mem | head
```

Solutions:

- Increase available memory
- Use checkpoint intervals to reduce memory usage
- Split large repositories into smaller ones
- For Docker, increase memory limits

5.5.7 Database Issues

SQLite Database Problems

Database Corruption

```
Error: Database disk image is malformed
```

Diagnosis:

```
# Check database integrity
sqlite3 data/borgitory.db "PRAGMA integrity_check;"
```

Solutions:

```
# Backup current database
cp data/borgitory.db data/borgitory.db.backup

# Attempt repair
sqlite3 data/borgitory.db ".recover" | sqlite3 data/borgitory_recovered.db

# Replace with recovered database
mv data/borgitory_recovered.db data/borgitory.db

# If repair fails, restore from backup
cp data/borgitory.db.backup data/borgitory.db
```

Database Lock Issues

```
Error: Database is locked
```

Solutions:

- Stop Borgitory application
- Check for zombie processes
- Remove lock files if present
- Restart application

5.5.8 Archive Browser Issues

FUSE Mount Problems

Archive Browser Disabled

```
Warning: Archive browsing disabled - FUSE not available
```

Solutions:

```
# For Docker deployment
docker run --cap-add SYS_ADMIN --device /dev/fuse ...

# For native installation, install FUSE
# Ubuntu/Debian:
sudo apt install fuse3
```

(continues on next page)

(continued from previous page)

```
# Check FUSE kernel module
lsmod | grep fuse
sudo modprobe fuse
```

Warning

Windows Users: FUSE is not available on Windows systems. Archive browsing functionality requires FUSE and will not work with PyPI installation on Windows.

Recommended Solution: Use Docker installation on Windows, which provides a Linux environment where FUSE works properly:

```
docker run -d \
  --name borgitory \
  -p 8000:8000 \
  --cap-add SYS_ADMIN \
  --device /dev/fuse \
  mlapaglia/borgitory:latest
```

Mount Permission Errors

```
Error: Permission denied mounting archive
```

Solutions:

- Ensure user has permission to use FUSE
- Add user to fuse group: `sudo usermod -a -G fuse $USER`
- Check `/etc/fuse.conf` allows user mounts
- Verify `user_allow_other` option is enabled

5.5.9 Web Interface Issues

Login Problems

Forgot Password

Solution:

```
# Reset admin password using CLI
borgitory reset-password admin

# Or recreate admin user
borgitory create-user admin --password newpassword --admin
```

Session Issues

```
Error: Session expired or invalid
```

Solutions:

- Clear browser cookies and cache
- Check system clock is correct

- Verify secret key hasn't changed
- Restart Borgitory service

Connection Refused

```
Error: Connection refused to localhost:8000
```

Diagnosis:

```
# Check if service is running
ps aux | grep borgitory

# Check port binding
netstat -tlnp | grep 8000

# For Docker
docker ps
docker logs borgitory
```

Solutions:

- Start Borgitory service
- Check port configuration
- Verify firewall allows port 8000
- Check if another service is using the port

5.5.10 Getting Help

Collecting Debug Information

When reporting issues, include:

System Information:

```
# Operating system
uname -a

# Python version
python --version

# Borgitory version
borgitory --version

# Docker version (if using Docker)
docker --version
```

Application Logs:

```
# For PyPI installation
borgitory serve --log-level debug

# For Docker
docker logs borgitory
```

(continues on next page)

(continued from previous page)

```
# Check system logs
journalctl -u borgitory
```

Configuration Details:

- Sanitized configuration (remove sensitive data)
- Repository setup information
- Cloud provider configuration (without credentials)
- Error messages with full stack traces

Support Channels

- **GitHub Issues**: https://github.com/mlapaglia/Borgitory/issues
- **GitHub Discussions**: https://github.com/mlapaglia/Borgitory/discussions
- **Documentation**: https://borgitory.readthedocs.io/

When reporting issues:

1. Search existing issues first
2. Provide detailed system information
3. Include relevant log entries
4. Describe steps to reproduce
5. Mention any recent changes to your setup

5.5.11 Common Error Codes

HTTP Error Codes

- **400 Bad Request**: Invalid input data or configuration
- **401 Unauthorized**: Authentication required or failed
- **403 Forbidden**: Insufficient permissions
- **404 Not Found**: Repository or resource doesn't exist
- **500 Internal Server Error**: Application error (check logs)

Borg Exit Codes

- **0**: Success
- **1**: Warning (operation succeeded with warnings)
- **2**: Error (operation failed)
- **3**: Interrupted (operation was interrupted)
- **4**: Repository does not exist
- **5**: Repository already exists

Rclone Exit Codes

- **0**: Success
- **1**: Syntax or usage error
- **2**: Error not otherwise categorised
- **3**: Directory not found
- **4**: File not found
- **5**: Temporary error (retry may help)
- **6**: Less serious errors (like file skipped)
- **7**: Fatal error (don't retry)
- **8**: Transfer exceeded - limit set by –max-transfer reached
- **9**: Operation successful, but no files transferred

5.5.12 Preventive Measures

Regular Maintenance

- **Monitor disk space** regularly
- **Test restore procedures** periodically
- **Update Borgitory** to latest versions
- **Backup configuration** and encryption keys
- **Monitor logs** for warnings and errors
- **Verify repository integrity** monthly

Best Practices

- Use strong, unique passphrases
- Implement 3-2-1 backup strategy
- Test backups before relying on them
- Monitor backup success rates
- Keep documentation updated
- Plan for disaster recovery scenarios

5.6 Adding New Cloud Providers

This document outlines the steps required to add support for a new cloud provider to the Borgitory cloud sync system.

5.6.1 Overview

The cloud sync system is designed with a modular architecture that makes adding new providers straightforward. Each provider consists of:

- **Storage Configuration Schema**: Defines and validates provider-specific settings
- **Storage Implementation**: Handles the actual upload/download operations via rclone
- **Frontend Templates**: Provides the user interface for configuration

- **Provider Registration**: Connects the provider to the main system via a decorator

Borgitory uses rclone for syncing. Borgitory can theoretically support any destination that rclone does.

5.6.2 Architecture

Each storage provider implements its own rclone integration directly. There is no centralized `RcloneService` – each storage class contains its own `sync_repository_to_{provider}()` and `test_{provider}_connection()` methods that build and execute rclone commands via the injected `CommandExecutorProtocol`.

The `CloudStorage` base class in `src/borgitory/services/cloud_providers/storage/base.py` provides:

- Abstract methods that every provider must implement (`upload_repository`, `test_connection`, `get_connection_info`, `get_sensitive_fields`, `get_display_details`)
- `_merge_async_generators()`: Concurrently merges stdout/stderr async generators using `asyncio.Queue` (avoids deadlocks from sequential reads)
- `parse_rclone_progress()`: Parses rclone progress output lines into structured data

These shared methods are inherited by all providers – do not duplicate them.

5.6.3 Registry Pattern

Borgitory uses a **registry pattern** for cloud providers, which means:

- **No hardcoded provider lists**: Providers are automatically discovered
- **Dynamic registration**: Use the `@register_provider` decorator to register your provider
- **Automatic integration**: Once registered, your provider appears in APIs, validation, and frontend dropdowns
- **Metadata support**: Include provider capabilities like versioning support, encryption, etc.

This eliminates the need to manually update multiple files when adding a provider.

5.6.4 Step-by-Step Implementation Guide

Step 1: Create the Storage Module

Create a new file: `src/borgitory/services/cloud_providers/storage/{provider_name}_storage.py`

This file contains three things: the config class, the storage class, and the provider registration.

Configuration class – extends `CloudStorageConfig` with provider-specific fields and validators:

```
import asyncio
from typing import AsyncGenerator, Callable, Dict, List, Optional, cast
from pydantic import Field, field_validator

from borgitory.protocols.command_executor_protocol import CommandExecutorProtocol
from borgitory.protocols.file_protocols import FileServiceProtocol
from borgitory.services.rclone_types import ConnectionTestResult, ProgressData

from .base import CloudStorage, CloudStorageConfig
from ..types import SyncEvent, SyncEventType, ConnectionInfo
from ..registry import register_provider, RcloneMethodMapping


class {ProviderName}StorageConfig(CloudStorageConfig):
    """Configuration for {Provider Name} storage"""
```

(continues on next page)

(continued from previous page)

```
    # Define your provider-specific fields here
    endpoint_url: str = Field(..., min_length=1, description="Provider endpoint URL")
    api_key: str = Field(..., min_length=1, description="API key for authentication")
    bucket_name: str = Field(..., min_length=1, max_length=255, description="Bucket/
↪container name")
    region: Optional[str] = Field(default=None, description="Region (if applicable)")

    @field_validator("endpoint_url")
    @classmethod
    def validate_endpoint_url(cls, v: str) -> str:
        if not v.startswith(("http://", "https://")):
            raise ValueError("Endpoint URL must start with http:// or https://")
        return v

    @field_validator("api_key")
    @classmethod
    def validate_api_key(cls, v: str) -> str:
        if len(v) < 10:
            raise ValueError("API key must be at least 10 characters long")
        return v
```

Storage class – implements all abstract methods from `CloudStorage` and contains the rclone integration:

```
class {ProviderName}Storage(CloudStorage):
    """
    {Provider Name} cloud storage implementation.
    """

    def __init__(
        self,
        config: {ProviderName}StorageConfig,
        command_executor: CommandExecutorProtocol,
        file_service: FileServiceProtocol,
    ) -> None:
        self._config = config
        self._command_executor = command_executor
        self._file_service = file_service

    def _build_{provider_name}_flags(self) -> List[str]:
        """Build provider-specific rclone flags."""
        return [
            "--{provider_name}-endpoint", self._config.endpoint_url,
            "--{provider_name}-api-key", self._config.api_key,
        ]

    async def upload_repository(
        self,
        repository_path: str,
        remote_path: str,
        progress_callback: Optional[Callable[[SyncEvent], None]] = None,
    ) -> None:
```

(continues on next page)

(continued from previous page)

```
        if progress_callback:
            progress_callback(
                SyncEvent(
                    type=SyncEventType.STARTED,
                    message=f"Starting {Provider Name} upload to {self._config.bucket_
↪name}",
                )
            )

        try:
            async for progress in self.sync_repository_to_{provider_name}(
                repository_path=repository_path,
                path_prefix=remote_path,
            ):
                if not progress_callback:
                    continue

                progress_type = progress.get("type")
                if progress_type == "progress":
                    progress_callback(
                        SyncEvent(
                            type=SyncEventType.PROGRESS,
                            message=str(progress.get("message", "Uploading...")),
                            progress=float(progress.get("percentage", 0.0) or 0.0),
                        )
                    )
                elif progress_type == "log":
                    progress_callback(
                        SyncEvent(
                            type=SyncEventType.LOG,
                            message=str(progress.get("message", "")),
                        )
                    )
                elif progress_type == "error":
                    error_msg = str(progress.get("message", "Unknown error"))
                    progress_callback(
                        SyncEvent(
                            type=SyncEventType.ERROR,
                            message=error_msg,
                            error=error_msg,
                        )
                    )
                    raise Exception(error_msg)
                elif progress_type == "completed":
                    if progress.get("status") != "success":
                        error_msg = f"{Provider Name} sync failed with return code
↪{progress.get('return_code')}"
                        progress_callback(
                            SyncEvent(
                                type=SyncEventType.ERROR,
                                message=error_msg,
                                error=error_msg,
```

(continues on next page)

(continued from previous page)

```
                        )
                    )
                    raise Exception(error_msg)

            if progress_callback:
                progress_callback(
                    SyncEvent(
                        type=SyncEventType.COMPLETED,
                        message="{Provider Name} upload completed successfully",
                    )
                )

        except Exception as e:
            error_msg = f"{Provider Name} upload failed: {str(e)}"
            if progress_callback:
                progress_callback(
                    SyncEvent(type=SyncEventType.ERROR, message=error_msg, error=str(e))
                )
            raise Exception(error_msg) from e

    async def test_connection(self) -> bool:
        try:
            result = await self.test_{provider_name}_connection()
            return result.get("status") == "success"
        except Exception:
            return False

    def get_connection_info(self) -> ConnectionInfo:
        return ConnectionInfo(
            provider="{provider_name}",
            details={
                "endpoint": self._config.endpoint_url,
                "bucket": self._config.bucket_name,
                "region": self._config.region or "default",
                "api_key": "***",
            },
        )

    def get_sensitive_fields(self) -> list[str]:
        return ["api_key"]

    def get_display_details(self, config_dict: Dict[str, object]) -> Dict[str, object]:
        endpoint = config_dict.get("endpoint_url", "Unknown")
        bucket = config_dict.get("bucket_name", "Unknown")
        region = config_dict.get("region", "default")

        provider_details = f"""
            <div><strong>Endpoint:</strong> {endpoint}</div>
            <div><strong>Bucket:</strong> {bucket}</div>
            <div><strong>Region:</strong> {region}</div>
        """.strip()
```

(continues on next page)

(continued from previous page)

```
        return {
            "provider_name": "{Provider Display Name}",
            "provider_details": provider_details,
        }

    @classmethod
    def get_rclone_mapping(cls) -> RcloneMethodMapping:
        return RcloneMethodMapping(
            sync_method="sync_repository_to_{provider_name}",
            test_method="test_{provider_name}_connection",
            parameter_mapping={
                "api_key": "api_key",
                "bucket_name": "bucket_name",
                "region": "region",
                "endpoint_url": "endpoint_url",
            },
            required_params=["repository", "api_key", "bucket_name"],
            optional_params={"region": None, "path_prefix": ""},
        )

    async def sync_repository_to_{provider_name}(
        self,
        repository_path: str,
        path_prefix: str = "",
    ) -> AsyncGenerator[ProgressData, None]:
        remote_path = f":{provider_name}:{self._config.bucket_name}"
        if path_prefix:
            remote_path = f"{remote_path}/{path_prefix}"

        command = [
            "rclone", "sync",
            repository_path, remote_path,
            "--progress", "--stats", "1s", "--verbose",
        ]
        command.extend(self._build_{provider_name}_flags())

        try:
            process = await self._command_executor.create_subprocess(
                command=command,
                stdout=asyncio.subprocess.PIPE,
                stderr=asyncio.subprocess.PIPE,
            )

            # Mask sensitive flags before yielding started event
            safe_parts = []
            skip_next = False
            sensitive_prefixes = ("--{provider_name}-api-key",)
            for part in command:
                if skip_next:
                    safe_parts.append("***")
                    skip_next = False
                elif part in sensitive_prefixes:
```

(continues on next page)

(continued from previous page)

```
                    safe_parts.append(part)
                    skip_next = True
                else:
                    safe_parts.append(part)

            yield cast(
                ProgressData,
                {"type": "started", "command": " ".join(safe_parts), "pid": process.pid},
            )

            async def read_stream(
                stream: Optional[asyncio.StreamReader], stream_type: str
            ) -> AsyncGenerator[ProgressData, None]:
                if stream is None:
                    return
                while True:
                    line = await stream.readline()
                    if not line:
                        break
                    decoded_line = line.decode("utf-8").strip()
                    progress_data = self.parse_rclone_progress(decoded_line)
                    if progress_data:
                        yield cast(ProgressData, {"type": "progress", **progress_data})
                    else:
                        yield cast(
                            ProgressData,
                            {"type": "log", "stream": stream_type, "message": decoded_
↪line},
                        )

            # _merge_async_generators is inherited from CloudStorage base class
            async for item in self._merge_async_generators(
                read_stream(process.stdout, "stdout"),
                read_stream(process.stderr, "stderr"),
            ):
                yield item

            return_code = await process.wait()
            yield cast(
                ProgressData,
                {
                    "type": "completed",
                    "return_code": return_code,
                    "status": "success" if return_code == 0 else "failed",
                },
            )

        except Exception as e:
            yield cast(ProgressData, {"type": "error", "message": str(e)})

    async def test_{provider_name}_connection(self) -> ConnectionTestResult:
        command = [
```

(continues on next page)

(continued from previous page)

```
            "rclone", "lsd",
            f":{provider_name}:{self._config.bucket_name}",
            "--max-depth", "1", "--verbose",
        ]
        command.extend(self._build_{provider_name}_flags())

        try:
            result = await self._command_executor.execute_command(
                command=command, timeout=30.0,
            )
            if result.success:
                return {"status": "success", "message": "Connection successful", "output
↪": result.stdout}
            return {"status": "failed", "message": result.stderr or "Connection failed"}
        except Exception as e:
            return {"status": "error", "message": str(e)}
```

Provider registration – at the bottom of the same file, add the `@register_provider` decorator:

```
@register_provider(
    name="{provider_name}",
    label="{Provider Display Name}",
    description="{Provider description}",
    supports_encryption=True,
    supports_versioning=False,
    requires_credentials=True,
)
class {ProviderName}Provider:
    """{Provider Name} provider registration"""
    config_class = {ProviderName}StorageConfig
    storage_class = {ProviderName}Storage
```

Step 2: Update the Storage Module Exports

Edit `src/borgitory/services/cloud_providers/storage/__init__.py`:

```
from .{provider_name}_storage import (
    {ProviderName}Storage,
    {ProviderName}StorageConfig,
    {ProviderName}Provider,
)  # Import provider to trigger registration

__all__ = [
    # ... existing exports ...
    "{ProviderName}Storage",
    "{ProviderName}StorageConfig",
    "{ProviderName}Provider",
]
```

Step 3: Register Sensitive Fields

Add your provider's sensitive fields to the static mapping in `src/borgitory/services/cloud_providers/cloud_sync_service.py`:

```
# In StorageFactory.get_sensitive_fields(), add to the mapping dict:
mapping = {
    "s3": ["access_key", "secret_key"],
    "sftp": ["password", "private_key"],
    "smb": ["pass"],
    "pcloud": ["token"],
    "{provider_name}": ["api_key"],  # <-- add your provider
}
```

This mapping is used when decrypting configs from the database. It must match what your storage class returns from `get_sensitive_fields()`.

Step 4: Create Frontend Template

Create `src/borgitory/templates/partials/cloud_sync/providers/{provider_name}/{provider_name}_fields.html`:

Templates must support both **create** and **edit** mode. Use `{% if config %}` to populate fields from existing config during edits. For sensitive fields, show a "leave empty to keep current" hint in edit mode.

```
<!-- {Provider Name} Fields -->
<div id="{provider_name}-fields" class="space-y-4">
    <div>
        <label class="block text-sm font-medium text-gray-900 dark:text-gray-100">
            Endpoint URL
        </label>
        <input type="text"
               name="provider_config[endpoint_url]"
               {% if config %}value="{{ config.endpoint_url or '' }}"{% endif %}
               placeholder="https://api.provider.com"
               class="input-modern mt-1">
    </div>
    <div>
        <label class="block text-sm font-medium text-gray-900 dark:text-gray-100">
            API Key
        </label>
        <input type="password"
               name="provider_config[api_key]"
               {% if config %}placeholder="Current value will be used if left empty"{%
↪endif %}
               class="input-modern mt-1">
    </div>
    <div>
        <label class="block text-sm font-medium text-gray-900 dark:text-gray-100">
            Bucket Name
        </label>
        <input type="text"
               name="provider_config[bucket_name]"
               {% if config %}value="{{ config.bucket_name or '' }}"{% endif %}
               placeholder="my-backup-bucket"
```

(continues on next page)

(continued from previous page)

```
                class="input-modern mt-1">
    </div>
    <div>
        <label class="block text-sm font-medium text-gray-900 dark:text-gray-100">
            Region (optional)
        </label>
        <input type="text"
               name="provider_config[region]"
               {% if config %}value="{{ config.region or '' }}"{% endif %}
               placeholder="us-east-1"
               class="input-modern mt-1">
    </div>
    <div>
        <label class="block text-sm font-medium text-gray-900 dark:text-gray-100">
            Path Prefix (optional)
        </label>
        <input type="text"
               name="path_prefix"
               {% if config %}value="{{ config.path_prefix or '' }}"{% endif %}
               placeholder="backups/borgitory"
               class="input-modern mt-1">
    </div>
</div>
```

Step 5: Template Integration (Automatic)

With the current implementation, templates are automatically discovered by checking if the template file exists on the filesystem. You don't need to manually update any API context variables.

The system will automatically:

- Check if the template file exists
- Include it in the provider fields if found
- Generate submit button text from registry metadata
- Handle provider validation through the registry

Simply create your template file and it will be automatically integrated.

Step 6: Create Tests

Create `tests/cloud_providers/test_{provider_name}_storage.py`:

Note: The project uses `asyncio_mode = "auto"` so you do not need `@pytest.mark.asyncio` on test methods.

```
import importlib

import pytest
from unittest.mock import AsyncMock

from borgitory.protocols.file_protocols import FileServiceProtocol
from borgitory.services.cloud_providers.storage.{provider_name}_storage import (
    {ProviderName}StorageConfig,
    {ProviderName}Storage,
```

(continues on next page)

(continued from previous page)

```
)
from borgitory.services.cloud_providers.types import SyncEvent
from borgitory.services.cloud_providers.registry import (
    clear_registry,
    get_supported_providers,
    get_provider_info,
)


class Test{ProviderName}StorageConfig:
    """Test {Provider Name} storage configuration validation"""

    def test_valid_config(self) -> None:
        config = {ProviderName}StorageConfig(
            endpoint_url="https://api.provider.com",
            api_key="valid-api-key-12345",
            bucket_name="test-bucket",
            region="us-east-1",
        )
        assert config.endpoint_url == "https://api.provider.com"
        assert config.bucket_name == "test-bucket"

    def test_invalid_endpoint_url(self) -> None:
        with pytest.raises(ValueError, match="Endpoint URL must start with"):
            {ProviderName}StorageConfig(
                endpoint_url="invalid-url",
                api_key="valid-api-key-12345",
                bucket_name="test-bucket",
            )

    def test_invalid_api_key(self) -> None:
        with pytest.raises(ValueError, match="API key must be at least"):
            {ProviderName}StorageConfig(
                endpoint_url="https://api.provider.com",
                api_key="short",
                bucket_name="test-bucket",
            )


class Test{ProviderName}Storage:
    """Test {Provider Name} storage implementation"""

    @pytest.fixture
    def mock_command_executor(self) -> AsyncMock:
        return AsyncMock()

    @pytest.fixture
    def mock_file_service(self) -> AsyncMock:
        return AsyncMock(spec=FileServiceProtocol)

    @pytest.fixture
    def storage_config(self) -> {ProviderName}StorageConfig:
```

(continues on next page)

(continued from previous page)

```
        return {ProviderName}StorageConfig(
            endpoint_url="https://api.provider.com",
            api_key="valid-api-key-12345",
            bucket_name="test-bucket",
            region="us-east-1",
        )

    @pytest.fixture
    def storage(
        self,
        storage_config: {ProviderName}StorageConfig,
        mock_command_executor: AsyncMock,
        mock_file_service: AsyncMock,
    ) -> {ProviderName}Storage:
        return {ProviderName}Storage(
            storage_config, mock_command_executor, mock_file_service
        )

    async def test_test_connection_success(
        self, storage: {ProviderName}Storage, mock_command_executor: AsyncMock
    ) -> None:
        mock_result = AsyncMock()
        mock_result.success = True
        mock_result.stdout = "test output"
        mock_result.stderr = ""
        mock_command_executor.execute_command.return_value = mock_result

        result = await storage.test_connection()
        assert result is True

    async def test_test_connection_failure(
        self, storage: {ProviderName}Storage, mock_command_executor: AsyncMock
    ) -> None:
        mock_command_executor.execute_command.side_effect = Exception(
            "Connection failed"
        )
        result = await storage.test_connection()
        assert result is False

    def test_get_sensitive_fields(self, storage: {ProviderName}Storage) -> None:
        sensitive_fields = storage.get_sensitive_fields()
        assert "api_key" in sensitive_fields

    def test_get_connection_info(self, storage: {ProviderName}Storage) -> None:
        info = storage.get_connection_info()
        assert info.provider == "{provider_name}"
        assert info.details["api_key"] == "***"


class Test{ProviderName}Registry:
    """Test {Provider Name} provider registration"""
```

(continues on next page)

(continued from previous page)

```
    @pytest.fixture(autouse=True)
    def fresh_registry(self) -> None:
        clear_registry()
        import borgitory.services.cloud_providers.storage.{provider_name}_storage

        importlib.reload(
            borgitory.services.cloud_providers.storage.{provider_name}_storage
        )

    def test_provider_in_supported_providers(self) -> None:
        providers = get_supported_providers()
        assert "{provider_name}" in providers

    def test_provider_info(self) -> None:
        info = get_provider_info("{provider_name}")
        assert info is not None
        assert info.name == "{provider_name}"
```

5.6.5 Important Implementation Notes

Credential Masking

Never leak sensitive credentials in `get_connection_info()`. Fully mask secrets with `"***"` rather than showing partial values like first/last N characters. For structured tokens (like OAuth JSON blobs), parse the structure and only expose non-sensitive metadata:

```
# Bad -- leaks partial credentials
"api_key": f"{self._config.api_key[:4]}***{self._config.api_key[-4:]}"

# Good -- fully masked
"api_key": "***"

# Good -- for structured tokens, show only non-sensitive fields
token_data = json.loads(self._config.token)
masked = {
    "token_type": token_data.get("token_type", "unknown"),
    "access_token": "***",
}
```

Similarly, when yielding the `started` event in your sync method, mask credential values in the command string before yielding.

Error Event Handling

The `upload_repository()` method must handle **all** progress event types from the sync generator:

- `progress`: Forward percentage/message to the callback
- `log`: Forward log messages to the callback
- `error`: Emit an ERROR SyncEvent **and raise** an exception
- `completed`: Check the `status` field – if not `"success"`, emit ERROR and raise

If you only check `completed` and `log`, an `error` event from the sync generator will be silently ignored and the upload will be treated as successful.

Shared Base Class Methods

The `CloudStorage` base class provides `_merge_async_generators()` and `parse_rclone_progress()` as concrete methods. Do **not** re-implement these in your provider. They are inherited automatically.

`_merge_async_generators()` uses `asyncio.Queue` with concurrent tasks to truly interleave stdout and stderr output. A sequential implementation (draining one stream then the other) risks deadlocks when a subprocess pipe buffer fills while the other stream is being read.

Registry Test Isolation

Other test modules may call `clear_registry()` which removes all registered providers. If your registry tests just import the provider module, Python's import cache means the `@register_provider` decorator won't re-execute. Always use a fixture that calls `clear_registry()` followed by `importlib.reload()` on your storage module:

```
@pytest.fixture(autouse=True)
def fresh_registry(self) -> None:
    clear_registry()
    import borgitory.services.cloud_providers.storage.{provider_name}_storage
    importlib.reload(borgitory.services.cloud_providers.storage.{provider_name}_storage)
```

Also add your provider to the `clean_registry` fixture in `tests/cloud_providers/test_provider_validation.py` so it's reloaded after registry clears in that file.

5.6.6 What's Simplified by the Registry Pattern

Thanks to the registry pattern, many things are automated:

- **Provider Discovery**: Automatic detection via `@register_provider` decorator
- **API Integration**: Providers appear in `/api/cloud-sync/providers` automatically
- **Template Discovery**: Automatic filesystem-based template detection
- **Submit Button Text**: Generated from registry metadata
- **Configuration Validation**: Uses registered config classes
- **Sensitive Field Detection**: Uses storage class methods

Note: Some manual steps are still required when adding new providers:

- Create template files manually
- Add the sensitive fields mapping in `cloud_sync_service.py`
- Update this documentation with provider-specific details

5.6.7 Enhanced Rclone Integration Pattern

RcloneMethodMapping

Each provider can define how its configuration maps to rclone method parameters:

```
from borgitory.services.cloud_providers.registry import RcloneMethodMapping

mapping = RcloneMethodMapping(
    sync_method="sync_repository_to_s3",
    test_method="test_s3_connection",
    parameter_mapping={
```

(continues on next page)

(continued from previous page)

```
        "access_key": "access_key_id",
        "secret_key": "secret_access_key",
        "bucket_name": "bucket_name",
        "region": "region",
    },
    required_params=["repository", "access_key_id", "secret_access_key", "bucket_name"],
    optional_params={"region": "us-east-1", "path_prefix": ""},
)
```

Two Ways to Define Rclone Mapping

Option 1: In Registration Decorator

```
@register_provider(
    name="myprovider",
    label="My Provider",
    rclone_mapping=mapping,
)
class MyProvider:
    config_class = MyProviderConfig
    storage_class = MyProviderStorage
```

Option 2: Auto-Discovery from Storage Class (preferred)

```
class MyProviderStorage(CloudStorage):
    @classmethod
    def get_rclone_mapping(cls) -> RcloneMethodMapping:
        return RcloneMethodMapping(
            sync_method="sync_repository_to_myprovider",
            test_method="test_myprovider_connection",
            parameter_mapping={"field": "param"},
            required_params=["repository", "param"],
        )

@register_provider(name="myprovider", label="My Provider")
class MyProvider:
    config_class = MyProviderConfig
    storage_class = MyProviderStorage
```

5.6.8 Checklist

When adding a new provider, verify:

1. `src/borgitory/services/cloud_providers/storage/{provider_name}_storage.py` created with config, storage, and registration classes
2. `src/borgitory/services/cloud_providers/storage/__init__.py` updated with imports and `__all__` entries
3. Sensitive fields mapping added to `src/borgitory/services/cloud_providers/cloud_sync_service.py`
4. Template created at `src/borgitory/templates/partials/cloud_sync/providers/{provider_name}/{provider_name}_fields.html`

5. Template supports both create and edit mode (`{% if config %}` bindings)
6. Tests created at `tests/cloud_providers/test_{provider_name}_storage.py`
7. Registry tests use `clear_registry()` + `importlib.reload()` fixture
8. Provider added to `clean_registry` fixture in `tests/cloud_providers/test_provider_validation.py`
9. `get_connection_info()` fully masks all credentials
10. Sync method masks sensitive flags in the started event command string
11. `upload_repository()` handles all event types: `progress`, `log`, `error`, `completed`

5.6.9 Provider-Specific Considerations

For Object Storage Providers (S3-like)

- Follow S3 patterns for bucket naming, regions, storage classes
- Consider implementing storage class options if supported
- Add endpoint URL validation for custom S3-compatible services

For File Transfer Providers (SFTP-like)

- Focus on connection authentication (keys, passwords, certificates)
- Validate host/port combinations
- Consider connection timeout and retry logic

For API-based Providers

- Implement proper API key/token validation and formatting
- Add rate limiting considerations
- Handle API versioning if applicable

5.6.10 Current Providers

Provider	Registration Name	Key Config Fields
S3	`s3`	`access_key`, `secret_key`, `bucket_name`, `region`, `endpoint_url`, `storage_class`, `provider_type`
SFTP	`sftp`	`host`, `username`, `port`, `password`, `private_key`, `remote_path`
SMB	`smb`	`host`, `user`, `pass`, `port`, `domain`, `share_name`
pCloud	`pcloud`	`token` (OAuth JSON), `hostname`, `root_folder_id`

5.7 API Reference

Borgitory provides a comprehensive REST API for all functionality, allowing you to integrate backup operations into your own scripts and applications.

5.7.1 Interactive Documentation

The API includes automatically generated interactive documentation:

- **Swagger UI**: Available at `http://localhost:8000/docs`
- **ReDoc**: Available at `http://localhost:8000/redoc`

These interfaces allow you to explore endpoints, view request/response schemas, and test API calls directly from your browser.

5.7.2 Authentication

Currently, the API uses session-based authentication through the web interface. API key authentication is planned for future releases.

Session Authentication:

1. Log in through the web interface at `http://localhost:8000`
2. Use the same browser session to make API calls
3. Include cookies in your requests for authentication

5.7.3 Base URL

All API endpoints are prefixed with the base URL of your Borgitory instance:

```
http://localhost:8000/api/
```

5.7.4 API Endpoints Overview

Repository Management

List Repositories

```
GET /api/repositories
```

Returns a list of all configured repositories.

Create Repository

```
POST /api/repositories
Content-Type: application/json

{
  "name": "My Backup Repo",
  "path": "/mnt/repos/my-backup-repo",
  "passphrase": "secure-passphrase"
}
```

Get Repository Details

```
GET /api/repositories/{repository_id}
```

Update Repository

```
PUT /api/repositories/{repository_id}
Content-Type: application/json

{
  "name": "Updated Repo Name",
  "path": "/mnt/repos/updated-path"
}
```

Delete Repository

```
DELETE /api/repositories/{repository_id}
```

Test Repository Connection

```
POST /api/repositories/{repository_id}/test
```

Backup Operations

Create Manual Backup

```
POST /api/backups/manual
Content-Type: application/json

{
  "repository_id": 1,
  "source_path": "/mnt/backup/sources/documents",
  "archive_name": "documents-{now:%Y%m%d-%H%M%S}",
  "compression": "lz4",
  "exclude_patterns": ["*.tmp", "*.log"]
}
```

List Archives

```
GET /api/repositories/{repository_id}/archives
```

Get Archive Details

```
GET /api/repositories/{repository_id}/archives/{archive_name}
```

Delete Archive

```
DELETE /api/repositories/{repository_id}/archives/{archive_name}
```

Browse Archive Contents

```
GET /api/repositories/{repository_id}/archives/{archive_name}/browse?path=/optional/
↪subpath
```

Download File from Archive

```
GET /api/repositories/{repository_id}/archives/{archive_name}/download?path=/path/to/file
```

Schedule Management

List Schedules

```
GET /api/schedules
```

Create Schedule

```
POST /api/schedules
Content-Type: application/json

{
  "name": "Daily Documents Backup",
  "repository_id": 1,
  "source_path": "/mnt/backup/sources/documents",
  "cron_expression": "0 2 * * *",
  "enabled": true,
  "compression": "lz4",
  "pruning_policy_id": 1,
  "cloud_sync_config_id": 1,
  "pre_job_hooks": "[{\"name\":\"Database Dump\",\"command\":\"pg_dump myapp > /tmp/dump.
↪sql\",\"critical\":true}]",
  "post_job_hooks": "[{\"name\":\"Cleanup\",\"command\":\"rm /tmp/dump.sql\",\"run_on_
↪job_failure\":true}]"
}
```

Update Schedule

```
PUT /api/schedules/{schedule_id}
```

Delete Schedule

```
DELETE /api/schedules/{schedule_id}
```

Enable/Disable Schedule

```
POST /api/schedules/{schedule_id}/toggle
```

Job Management

List Jobs

```
GET /api/jobs?status=running&limit=50&offset=0
```

Get Job Details

```
GET /api/jobs/{job_id}
```

Cancel Job

```
POST /api/jobs/{job_id}/cancel
```

Retry Failed Job

```
POST /api/jobs/{job_id}/retry
```

Get Job Logs

```
GET /api/jobs/{job_id}/logs
```

Stream Job Progress

```
GET /api/jobs/{job_id}/stream
```

Returns Server-Sent Events for real-time job progress updates.

Cloud Sync

List Cloud Configurations

```
GET /api/cloud-sync/configs
```

Create Cloud Configuration

```
POST /api/cloud-sync/configs
Content-Type: application/json

{
  "name": "My S3 Backup",
  "provider": "s3",
  "provider_config": {
    "access_key_id": "AKIAIOSFODNN7EXAMPLE",
    "secret_access_key": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
    "bucket_name": "my-backup-bucket",
    "region": "us-east-1"
  },
  "path_prefix": "borgitory-backups/"
}
```

Test Cloud Connection

```
POST /api/cloud-sync/configs/{config_id}/test
```

Manual Cloud Sync

```
POST /api/cloud-sync/sync
Content-Type: application/json

{
  "repository_id": 1,
  "cloud_sync_config_id": 1
}
```

List Supported Providers

```
GET /api/cloud-sync/providers
```

5.7.5 Response Formats

Standard Response Structure

All API responses follow a consistent structure:

Success Response:

```
{
  "status": "success",
  "data": {
    // Response data here
  },
  "message": "Operation completed successfully"
}
```

Error Response:

```
{
  "status": "error",
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input data",
    "details": {
      "field": ["Field is required"]
    }
  }
}
```

HTTP Status Codes

- **200 OK**: Successful GET, PUT requests
- **201 Created**: Successful POST requests that create resources
- **204 No Content**: Successful DELETE requests
- **400 Bad Request**: Invalid request data or parameters
- **401 Unauthorized**: Authentication required
- **403 Forbidden**: Insufficient permissions
- **404 Not Found**: Resource not found
- **409 Conflict**: Resource already exists or conflict with current state
- **422 Unprocessable Entity**: Validation errors
- **500 Internal Server Error**: Server-side errors

5.7.6 Example Usage

Python Example

```
import requests
import json

# Base configuration
```

(continues on next page)

(continued from previous page)

```
BASE_URL = "http://localhost:8000/api"
session = requests.Session()

# Login first (through web interface)
# Then use the same session for API calls

def create_repository():
    """Create a new repository"""
    data = {
        "name": "Documents Backup",
        "path": "/mnt/repos/documents",
        "passphrase": "secure-passphrase-123"
    }

    response = session.post(f"{BASE_URL}/repositories", json=data)
    if response.status_code == 201:
        repo = response.json()["data"]
        print(f"Created repository: {repo['name']} (ID: {repo['id']})")
        return repo
    else:
        print(f"Error: {response.json()}")

def start_backup(repository_id):
    """Start a manual backup"""
    data = {
        "repository_id": repository_id,
        "source_path": "/mnt/backup/sources/documents",
        "archive_name": "docs-{now:%Y%m%d-%H%M%S}",
        "compression": "lz4"
    }

    response = session.post(f"{BASE_URL}/backups/manual", json=data)
    if response.status_code == 201:
        job = response.json()["data"]
        print(f"Started backup job: {job['id']}")
        return job
    else:
        print(f"Error: {response.json()}")

# Usage
repo = create_repository()
if repo:
    job = start_backup(repo["id"])
```

JavaScript Example

```
// API client class
class BorgitoryAPI {
  constructor(baseUrl = 'http://localhost:8000/api') {
    this.baseUrl = baseUrl;
  }
```

(continues on next page)

(continued from previous page)

```
  async request(method, endpoint, data = null) {
    const config = {
      method,
      credentials: 'include', // Include cookies for session auth
      headers: {
        'Content-Type': 'application/json',
      },
    };

    if (data) {
      config.body = JSON.stringify(data);
    }

    const response = await fetch(`${this.baseUrl}${endpoint}`, config);
    const result = await response.json();

    if (!response.ok) {
      throw new Error(result.error?.message || 'API request failed');
    }

    return result.data;
  }

  // Repository methods
  async getRepositories() {
    return this.request('GET', '/repositories');
  }

  async createRepository(name, path, passphrase) {
    return this.request('POST', '/repositories', {
      name, path, passphrase
    });
  }

  // Backup methods
  async startManualBackup(repositoryId, sourcePath, archiveName, compression = 'lz4') {
    return this.request('POST', '/backups/manual', {
      repository_id: repositoryId,
      source_path: sourcePath,
      archive_name: archiveName,
      compression
    });
  }
}

// Usage example
const api = new BorgitoryAPI();

async function example() {
  try {
    // Create repository
```

(continues on next page)

(continued from previous page)

```
    const repo = await api.createRepository(
      'My Documents',
      '/mnt/repos/documents',
      'secure-passphrase'
    );
    console.log('Created repository:', repo);

    // Start backup
    const job = await api.startManualBackup(
      repo.id,
      '/mnt/backup/sources/documents',
      'docs-{now:%Y%m%d-%H%M%S}'
    );
    console.log('Started backup:', job);

  } catch (error) {
    console.error('API error:', error);
  }
}
```

5.7.7 Future API Enhancements

Planned improvements for future releases:

- **API Key Authentication**: Dedicated API keys for programmatic access
- **Webhooks**: HTTP callbacks for job completion and events
- **GraphQL Endpoint**: More flexible query capabilities
- **Bulk Operations**: Batch create/update/delete operations
- **Advanced Filtering**: More sophisticated query parameters
- **API Versioning**: Versioned endpoints for backward compatibility

For the most up-to-date API documentation and testing interface, always refer to the interactive documentation at `/docs` and `/redoc` endpoints of your running Borgitory instance.

5.8 Architecture Overview

This document provides a technical overview of Borgitory's architecture, design principles, and implementation details.

5.8.1 System Architecture

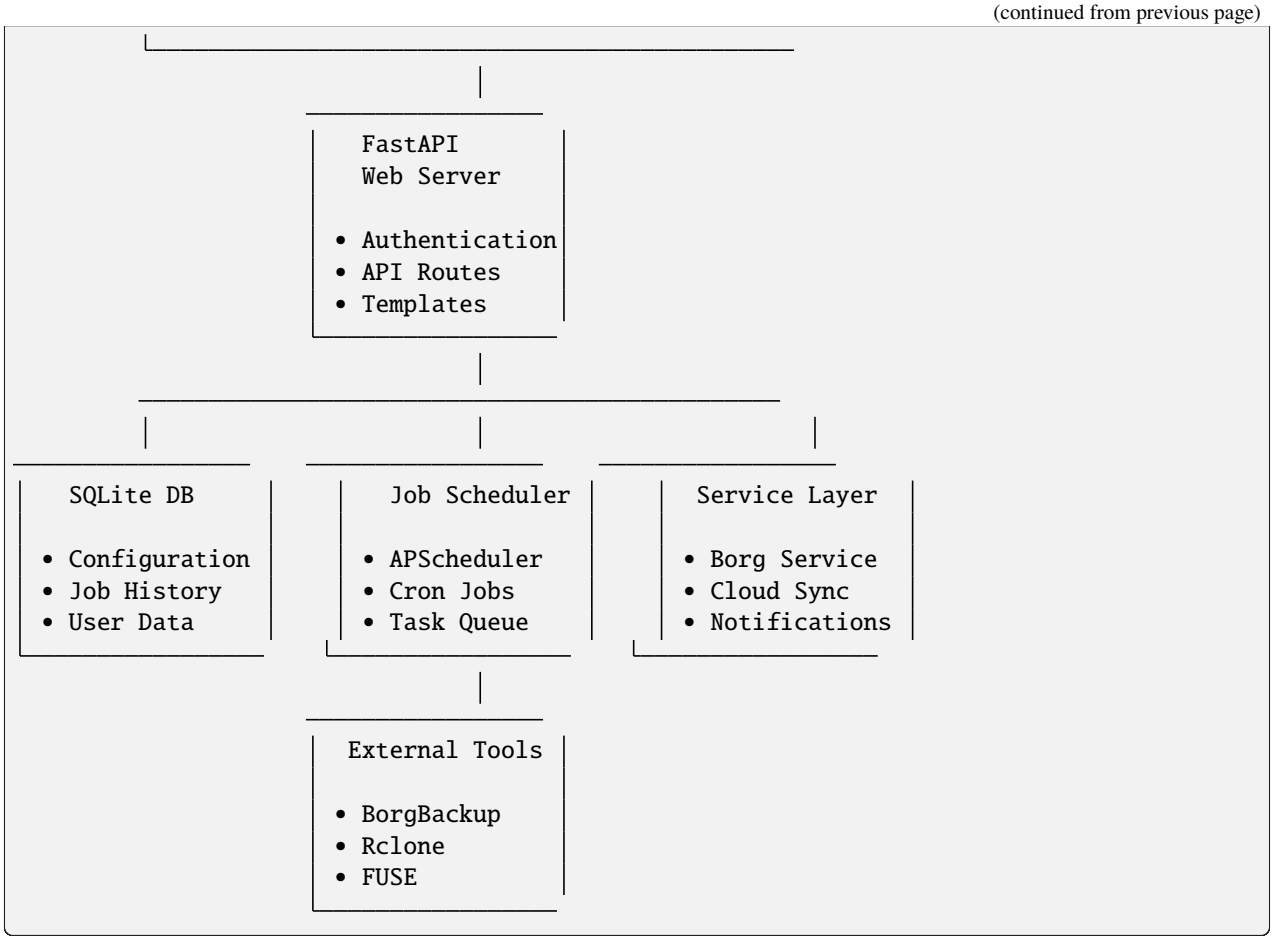
High-Level Overview

Borgitory follows a modern web application architecture with clear separation of concerns:

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Web Browser   │    │  Mobile Device  │    │   API Client    │
│                 │    │                 │    │                 │
│ HTMX + Alpine   │    │ Responsive UI   │    │ REST API        │
└─────────┬───────┘    └─────────┬───────┘    └─────────┬───────┘
          │                      │                      │
```

(continues on next page)

(continued from previous page)



5.8.2 Backend Architecture

FastAPI Application

Core Components:

- **Main Application** (`src/borgitory/main.py`)
  - FastAPI application instance
  - Middleware configuration
  - Static file serving
  - Template engine setup
- **API Routes** (`src/borgitory/api/`)
  - RESTful API endpoints
  - Request/response handling
  - Input validation
  - Error handling
- **Dependencies** (`src/borgitory/dependencies.py`)
  - Dependency injection container

- Service initialization
- Database session management
- Authentication dependencies

Service Layer Architecture

Service Pattern Implementation:

```
# Service interfaces define contracts
class BackupService:
    async def create_backup(self, config: BackupConfig) -> Job
    async def get_backup_status(self, job_id: str) -> JobStatus

# Concrete implementations handle business logic
class BorgBackupService(BackupService):
    def __init__(self, borg_service: BorgService):
        self._borg_service = borg_service

# Dependency injection wires everything together
@lru_cache()
def get_backup_service() -> BackupService:
    return BorgBackupService(get_borg_service())
```

Key Services:

- **BorgService** - BorgBackup command execution and management
- **CloudSyncService** - Multi-provider cloud synchronization
- **JobService** - Background job management and monitoring
- **SchedulerService** - Cron-based task scheduling
- **NotificationService** - Push notification handling
- **RecoveryService** - Archive browsing and file recovery

5.8.3 Database Layer

SQLAlchemy ORM

Model Architecture:

```
# Base model with common fields
class BaseModel:
    id: int
    created_at: datetime
    updated_at: datetime

# Domain models
class Repository(BaseModel):
    name: str
    path: str
    encrypted_passphrase: str

class Schedule(BaseModel):
```

(continues on next page)

(continued from previous page)

```
    name: str
    repository_id: int
    cron_expression: str
    enabled: bool

class Job(BaseModel):
    type: JobType
    status: JobStatus
    repository_id: int
    started_at: datetime
    completed_at: datetime
```

Database Features:

- **SQLite** for lightweight deployment
- **Alembic** for schema migrations
- **Connection pooling** for performance
- **Async support** throughout the stack
- **Encrypted sensitive data** storage

5.8.4 Frontend Architecture

Modern Web Stack

Technology Choices:

- **HTMX** - Dynamic HTML updates without JavaScript frameworks
- **Alpine.js** - Lightweight JavaScript reactivity
- **Tailwind CSS** - Utility-first styling with responsive design
- **Jinja2** - Server-side template rendering

Why This Stack:

1. **Simplicity** - Minimal JavaScript complexity
2. **Performance** - Fast page loads and updates
3. **SEO-Friendly** - Server-side rendered content
4. **Progressive Enhancement** - Works without JavaScript
5. **Developer Experience** - Familiar template-based approach

Template Architecture

Template Hierarchy:

```
templates/
├── base.html                    # Base layout
├── components/                  # Reusable components
    ├── navigation.html
    ├── job_progress.html
    └── repository_card.html
```

(continues on next page)

(continued from previous page)

```
├── pages/                      # Full page templates
│   ├── dashboard.html
│   ├── repositories.html
│   └── schedules.html
└── partials/                   # HTMX partial updates
    ├── job_list.html
    ├── backup_form.html
    └── cloud_sync/
        ├── provider_form.html
        └── providers/
            ├── s3_fields.html
            └── azure_fields.html
```

Template Features:

- **Component-based design** for reusability
- **Partial updates** via HTMX for dynamic behavior
- **Responsive design** with mobile-first approach
- **Dark/light theme** support
- **Accessibility** features built-in

5.8.5 Job Management System

Background Job Processing

Job Architecture:

```python
class Job:
    id: str
    type: JobType  # BACKUP, PRUNE, CLOUD_SYNC
    status: JobStatus  # PENDING, RUNNING, COMPLETED, FAILED
    repository_id: int
    configuration: dict
    progress: JobProgress
    tasks: List[Task]

class Task:
    id: str
    job_id: str
    name: str
    status: TaskStatus
    output: str
    started_at: datetime
    completed_at: datetime
```

Job Lifecycle:

1. **Creation** - Job created with PENDING status
2. **Queuing** - Added to scheduler queue
3. **Execution** - Status changes to RUNNING
4. **Progress Updates** - Real-time progress via SSE

5. **Task Management** - Individual tasks tracked
6. **Completion** - Final status (COMPLETED/FAILED)
7. **Cleanup** - Temporary resources cleaned up

Scheduling Integration:

```
from apscheduler.schedulers.asyncio import AsyncIOScheduler

class SchedulerService:
    def __init__(self):
        self.scheduler = AsyncIOScheduler()

    async def add_schedule(self, schedule: Schedule):
        self.scheduler.add_job(
            func=self.execute_backup,
            trigger='cron',
            **parse_cron_expression(schedule.cron_expression),
            id=f"schedule_{schedule.id}",
            args=[schedule.id]
        )
```

JobManager Dependency Injection Pattern

Overview

The JobManager uses a **dual-function dependency injection pattern** to handle both FastAPI request-scoped dependencies and application-scoped singleton access. This pattern is essential for managing long-running background tasks while maintaining proper dependency injection principles.

The Pattern

```
@lru_cache()
def get_job_manager_singleton() -> "JobManagerProtocol":
    """
    Create JobManager singleton for application-scoped use.

     Use for: Singletons, direct instantiation, tests, background tasks
     Don't use for: FastAPI endpoints (use get_job_manager_dependency instead)
    """
    # Resolve all dependencies directly (not via FastAPI DI)
    # ... dependency resolution ...
    return JobManager(config=config, dependencies=dependencies)

def get_job_manager_dependency(
    # All FastAPI dependencies listed here for injection
    config: JobManagerConfig = Depends(_create_job_manager_config),
    job_executor: JobExecutor = Depends(get_job_executor),
    # ... other dependencies ...
) -> "JobManagerProtocol":
    """
    Provide JobManager with FastAPI dependency injection.

     Use for: FastAPI endpoints with Depends(get_job_manager_dependency)
     Don't use for: Direct calls, background tasks, tests
```

(continues on next page)

(continued from previous page)

```
    """
    # Both functions return the same singleton instance
    return get_job_manager_singleton()
```

Why This Pattern is Necessary

The Problem: FastAPI + Long-Running Tasks

FastAPI's dependency injection system is designed for **request-scoped** operations. However, job management requires:

1. **State Persistence** - Jobs must persist across multiple HTTP requests
2. **Background Processing** - Tasks run independently of HTTP request lifecycle
3. **Singleton Behavior** - All parts of the application must see the same job state

The Challenge

```
#  BROKEN: This creates a new JobManager for each request
@router.post("/jobs/backup")
async def create_backup(job_manager: JobManager = Depends(get_job_manager)):
    job_id = job_manager.start_backup()  # Job stored in instance A
    return {"job_id": job_id}

@router.get("/jobs/{job_id}")
async def get_job(job_id: str, job_manager: JobManager = Depends(get_job_manager)):
    return job_manager.get_job(job_id)  # Looking in instance B - job not found!
```

The Solution: Dual Functions

The dual-function pattern ensures:

- **Same Instance** - Both functions return the identical singleton
- **Proper DI** - FastAPI endpoints get dependency injection
- **Direct Access** - Background tasks get direct singleton access

Usage Examples

FastAPI Endpoint Usage

```
from borgitory.dependencies import RequestScopedJobManager

@router.post("/jobs/backup")
async def create_backup(
    request: BackupRequest,
    job_manager: RequestScopedJobManager,  # Uses Depends() internally
) -> dict:
    """Create a backup job via FastAPI endpoint."""
    job_id = await job_manager.start_backup_job(
        repository_id=request.repository_id,
        source_path=request.source_path
    )
    return {"job_id": job_id, "status": "started"}

@router.get("/jobs/{job_id}")
async def get_job_status(
```

(continues on next page)

(continued from previous page)

```
    job_id: str,
    job_manager: RequestScopedJobManager,
) -> dict:
    """Get job status via FastAPI endpoint."""
    job = job_manager.get_job(job_id)
    if not job:
        raise HTTPException(status_code=404, detail="Job not found")

    return {
        "job_id": job_id,
        "status": job.status,
        "progress": job.progress
    }
```

Background Task Usage

```
import asyncio
from borgitory.dependencies import get_job_manager_singleton

async def cleanup_completed_jobs():
    """Background task to clean up old completed jobs."""
    # Direct singleton access - no FastAPI DI needed
    job_manager = get_job_manager_singleton()

    completed_jobs = job_manager.get_completed_jobs(older_than_days=7)
    for job in completed_jobs:
        await job_manager.cleanup_job(job.id)
        print(f"Cleaned up job {job.id}")

async def job_monitor_daemon():
    """Long-running daemon to monitor job health."""
    job_manager = get_job_manager_singleton()

    while True:
        # Check for stuck jobs
        stuck_jobs = job_manager.get_stuck_jobs()
        for job in stuck_jobs:
            await job_manager.restart_job(job.id)

        await asyncio.sleep(60)  # Check every minute

# Start background tasks
asyncio.create_task(cleanup_completed_jobs())
asyncio.create_task(job_monitor_daemon())
```

Testing Usage

```
import pytest
from borgitory.dependencies import get_job_manager_singleton

def test_job_creation():
    """Test job creation with direct singleton access."""
```

(continues on next page)

(continued from previous page)

```
    job_manager = get_job_manager_singleton()

    job_id = job_manager.create_job("backup", {"source": "/data"})
    assert job_id is not None

    job = job_manager.get_job(job_id)
    assert job.status == "pending"

@pytest.fixture
def job_manager():
    """Fixture providing JobManager for tests."""
    return get_job_manager_singleton()

def test_job_lifecycle(job_manager):
    """Test complete job lifecycle."""
    job_id = job_manager.start_backup_job("/data", "/backup")

    # Job should be running
    job = job_manager.get_job(job_id)
    assert job.status in ["pending", "running"]

    # Simulate completion
    job_manager.complete_job(job_id)
    job = job_manager.get_job(job_id)
    assert job.status == "completed"
```

Type Aliases for Clarity

```
# Semantic type aliases make usage intent crystal clear
ApplicationScopedJobManager = "JobManagerProtocol"  # Direct singleton access
RequestScopedJobManager = Annotated[
    "JobManagerProtocol",
    Depends(get_job_manager_dependency)
]  # FastAPI DI

# Usage examples:
def background_task():
    manager: ApplicationScopedJobManager = get_job_manager_singleton()

async def api_endpoint(manager: RequestScopedJobManager):
    # FastAPI automatically injects the singleton
    pass
```

Key Benefits

1. **State Consistency**

   ```
   # Same job visible across all contexts
   job_id = create_backup_via_api()  # FastAPI endpoint
   status = check_job_in_background(job_id)  # Background task
   assert status is not None  #  Works!
   ```

2. **Proper Dependency Injection**

```
# FastAPI endpoints get full DI benefits
async def endpoint(
    job_manager: RequestScopedJobManager,  # Injected
    db: Session = Depends(get_db),         # Injected
    user: User = Depends(get_current_user) # Injected
):
    # All dependencies properly resolved
```

3. **Performance Optimization**

```
# Singleton pattern avoids expensive re-initialization
@lru_cache()  # Cached after first call
def get_job_manager_singleton():
    # Heavy initialization only happens once
    return JobManager(expensive_setup=True)
```

4. **Testing Flexibility**

```
# Easy to mock in tests
def test_with_mock():
    with patch('borgitory.dependencies.get_job_manager_singleton') as mock:
        mock.return_value = MockJobManager()
        # Test uses mock instead of real singleton
```

Anti-Patterns to Avoid

Don't Mix the Functions

```
# WRONG: Using dependency function directly
def background_task():
    # This will fail - Depends objects can't be called directly
    manager = get_job_manager_dependency()  #  RuntimeError
```

Don't Create Multiple Instances

```
# WRONG: Creating JobManager directly
def some_function():
    manager = JobManager()  #  Creates separate instance
    # This instance won't see jobs from other parts of the app
```

Don't Use Global Variables

```
# WRONG: Module-level global
_job_manager = None

def get_job_manager():
    global _job_manager
    if not _job_manager:
        _job_manager = JobManager()  #  Anti-pattern
    return _job_manager
```

Implementation Details

Dependency Resolution

The singleton function resolves all dependencies directly:

```
@lru_cache()
def get_job_manager_singleton():
    # Direct dependency resolution (not via FastAPI)
    config = _create_job_manager_config()
    job_executor = get_job_executor()
    output_manager = get_job_output_manager()
    # ... resolve all dependencies ...

    return JobManager(config=config, dependencies=dependencies)
```

Runtime Safety

The dependency function includes runtime checks:

```
def get_job_manager_dependency(...):
    # Prevent misuse
    if hasattr(job_executor, "dependency"):
        raise RuntimeError(
            "get_job_manager_dependency() was called directly with Depends objects. "
            "Use get_job_manager_singleton() for direct calls instead."
        )

    return get_job_manager_singleton()
```

Pattern Summary

This dual-function pattern solves the fundamental challenge of using FastAPI's request-scoped dependency injection with application-scoped services that manage long-running tasks. It provides:

- **Consistency** - Same instance across all contexts
- **Proper DI** - Full FastAPI dependency injection support
- **Flexibility** - Works in endpoints, background tasks, and tests
- **Performance** - Singleton pattern with caching
- **Safety** - Runtime checks prevent misuse

The pattern is essential for any FastAPI application that needs to manage stateful, long-running operations while maintaining clean dependency injection architecture.

Real-Time Updates

Server-Sent Events (SSE):

```
from sse_starlette import EventSourceResponse

async def stream_job_progress(job_id: str):
    async def event_generator():
        while True:
            job = await get_job(job_id)
            if job.status in [JobStatus.COMPLETED, JobStatus.FAILED]:
                break

            yield {
                "event": "progress",
```

(continues on next page)

(continued from previous page)

```
                "data": json.dumps({
                    "job_id": job.id,
                    "status": job.status,
                    "progress": job.progress.percentage,
                    "current_task": job.current_task
                })
            }

            await asyncio.sleep(1)

    return EventSourceResponse(event_generator())
```

5.8.6 Cloud Provider System

Registry Pattern

Dynamic Provider Registration:

```
# Provider registry for dynamic discovery
PROVIDER_REGISTRY = {}

def register_provider(name: str, label: str, **metadata):
    def decorator(cls):
        PROVIDER_REGISTRY[name] = {
            'class': cls,
            'label': label,
            'metadata': metadata
        }
        return cls
    return decorator

@register_provider(
    name="s3",
    label="Amazon S3",
    supports_encryption=True,
    supports_versioning=True
)
class S3Provider:
    config_class = S3StorageConfig
    storage_class = S3Storage
```

Provider Interface:

```
class CloudStorage(ABC):
    @abstractmethod
    async def upload_repository(
        self,
        repository_path: str,
        remote_path: str,
        progress_callback: Optional[Callable] = None
    ) -> None:
        pass
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    async def test_connection(self) -> bool:
        pass

    @abstractmethod
    def get_sensitive_fields(self) -> List[str]:
        pass
```

Benefits of Registry Pattern:

- **No hardcoded provider lists** - Automatic discovery
- **Dynamic integration** - Providers auto-appear in UI
- **Extensibility** - Easy to add new providers
- **Type safety** - Compile-time validation
- **Metadata-driven** - UI behavior based on capabilities

5.8.7 Security Architecture

Authentication & Authorization

Current Implementation:

- **Session-based authentication** with secure cookies
- **BCrypt password hashing** with salt
- **CSRF protection** on all forms
- **Input validation** and sanitization
- **SQL injection prevention** via ORM

Security Features:

```
# Password hashing
from passlib.context import CryptContext
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# Credential encryption
from cryptography.fernet import Fernet
cipher_suite = Fernet(secret_key)
encrypted_password = cipher_suite.encrypt(password.encode())

# Session security
app.add_middleware(SessionMiddleware, secret_key=secret_key)
```

Planned Security Enhancements:

- **Multi-user support** with role-based access control
- **API key authentication** for programmatic access
- **OAuth integration** for external authentication
- **Audit logging** for security events

- **Rate limiting** and abuse prevention

Data Protection

Sensitive Data Handling:

- **Repository passphrases** encrypted with Fernet
- **Cloud provider credentials** encrypted at rest
- **API keys** masked in UI and logs
- **Database encryption** for sensitive fields
- **Secure key management** with rotation support

Backup Security:

- **End-to-end encryption** via BorgBackup
- **Client-side encryption** before cloud upload
- **Key derivation** from user passphrases
- **No plaintext secrets** in configuration files

5.8.8 Performance Considerations

Scalability Design

Current Optimizations:

- **Async/await** throughout the application
- **Connection pooling** for database access
- **Background job processing** for long-running tasks
- **Streaming responses** for large file downloads
- **Efficient FUSE mounting** for archive browsing

Performance Monitoring:

```
# Request timing middleware
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)
    return response
```

Scaling Strategies:

- **Horizontal scaling** with multiple instances
- **Load balancing** for high availability
- **Database sharding** for large deployments
- **CDN integration** for static assets
- **Caching layers** for frequently accessed data

Monitoring & Observability

Built-in Monitoring:

- **Health checks** for service availability
- **Job execution metrics** and history
- **Error tracking** and reporting
- **Performance profiling** for optimization
- **Resource usage monitoring**

Logging Architecture:

```
import structlog

logger = structlog.get_logger()

# Structured logging with context
logger.info(
    "backup_started",
    repository_id=repo.id,
    archive_name=archive_name,
    source_path=source_path
)
```

5.8.9 Deployment Architecture

Container Strategy

Docker Benefits:

- **Isolated environment** with all dependencies
- **Consistent deployment** across environments
- **Easy updates** and rollbacks
- **Resource management** and limits
- **Security isolation** from host system

Container Architecture:

```
FROM python:3.11-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
    borgbackup \
    rclone \
    fuse3 \
    && rm -rf /var/lib/apt/lists/*

# Copy application
COPY . /app
WORKDIR /app

# Install Python dependencies
```

(continues on next page)

(continued from previous page)

```
RUN pip install -e .

# Configure runtime
EXPOSE 8000
CMD ["borgitory", "serve"]
```

Volume Strategy:

- **Application data** - Persistent SQLite database and configuration
- **Repository storage** - Borg repository locations
- **Backup sources** - Read-only access to source directories
- **Temporary storage** - Scratch space for operations

5.8.10 Development Workflow

Code Organization

Project Structure:

```
src/borgitory/
├── __init__.py
├── main.py                     # FastAPI application
├── cli.py                      # Command-line interface
├── config.py                   # Configuration management
├── dependencies.py             # Dependency injection
├── api/                        # API route modules
│   ├── repositories.py
│   ├── jobs.py
│   └── cloud_sync.py
├── models/                     # SQLAlchemy models
│   ├── database.py
│   ├── repository.py
│   └── job.py
├── services/                   # Business logic services
│   ├── borg_service.py
│   ├── cloud_sync_service.py
│   └── job_service.py
├── templates/                  # Jinja2 templates
│   ├── base.html
│   ├── pages/
│   └── components/
└── utils/                      # Utility modules
    ├── security.py
    └── validation.py
```

Testing Strategy:

- **Unit tests** for individual components
- **Integration tests** for service interactions
- **API tests** for endpoint functionality
- **End-to-end tests** for complete workflows

- **Performance tests** for scalability validation

Quality Assurance:

```
# GitHub Actions workflow
- name: Lint code
  run: ruff check .

- name: Type checking
  run: mypy src/

- name: Run tests
  run: pytest --cov=borgitory

- name: Security scan
  run: bandit -r src/
```

5.8.11 Future Architecture Plans

Planned Improvements

Database Enhancements:

- **PostgreSQL support** for production deployments
- **Database clustering** for high availability
- **Read replicas** for scaling read operations
- **Connection pooling** optimization

Microservices Evolution:

- **Service separation** for independent scaling
- **Message queues** for inter-service communication
- **API gateway** for unified access
- **Service mesh** for advanced networking

Cloud-Native Features:

- **Kubernetes operator** for orchestration
- **Helm charts** for deployment
- **Prometheus metrics** for monitoring
- **Distributed tracing** for observability

Advanced Features:

- **Plugin system** for extensibility
- **Workflow engine** for complex operations
- **Multi-tenancy** support
- **Advanced analytics** and reporting

This architecture provides a solid foundation for current needs while remaining flexible enough to evolve with future requirements.

5.9 Changelog

All notable changes to Borgitory are documented here.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

5.9.1 [Unreleased]

Added

- Comprehensive documentation on Read the Docs
- Enhanced cloud provider system with registry pattern
- Multi-provider cloud sync support
- Advanced archive browsing with FUSE mounting
- Real-time job monitoring with Server-Sent Events

Changed

- Improved user interface with HTMX and Alpine.js
- Enhanced security with encrypted credential storage
- Better error handling and validation throughout
- Optimized Docker container with proper FUSE support

5.9.2 [1.0.0] - 2024-XX-XX

Added

- Initial stable release
- Web-based management interface for BorgBackup
- Repository management with multiple repository support
- Manual and scheduled backup operations
- Real-time progress monitoring
- Archive browser with file download capability
- Cloud synchronization with S3-compatible providers
- Push notifications via Pushover
- User authentication and session management
- RESTful API with OpenAPI documentation
- Docker deployment support
- PyPI package distribution

Features

- **Repository Management**
  - Add, configure, and manage multiple Borg repositories
  - Secure passphrase storage with encryption

  - Repository validation and connection testing
- **Backup Operations**
  - Manual backups with configurable compression
  - Real-time progress tracking
  - Archive naming with timestamp templates
  - Exclude patterns and file filtering
- **Scheduling System**
  - Cron-based backup scheduling
  - Human-readable cron descriptions
  - Schedule enable/disable functionality
  - Integration with pruning and notifications
- **Archive Management**
  - Interactive archive browser with FUSE mounting
  - Direct file downloads from archives
  - Archive pruning with retention policies
  - Simple and advanced pruning strategies
- **Cloud Integration**
  - Multi-provider cloud sync support
  - Amazon S3 and S3-compatible services
  - Automatic sync after backups
  - Real-time sync progress monitoring
- **Notifications**
  - Pushover integration for mobile alerts
  - Success and failure notifications
  - Per-schedule notification configuration
- **User Interface**
  - Modern responsive web interface
  - HTMX for dynamic updates
  - Alpine.js for interactivity
  - Tailwind CSS for styling
  - Mobile-friendly design
- **API & Integration**
  - Comprehensive REST API
  - OpenAPI/Swagger documentation
  - Real-time updates via Server-Sent Events
  - Session-based authentication

- **Deployment**
  - Docker containerization
  - PyPI package distribution
  - Docker Compose configurations
  - Volume mounting strategies

Technical Details

- **Backend**: FastAPI, SQLAlchemy, APScheduler
- **Frontend**: HTMX, Alpine.js, Tailwind CSS, Jinja2
- **Database**: SQLite with Alembic migrations
- **Security**: BCrypt password hashing, Fernet encryption
- **Dependencies**: BorgBackup, Rclone, FUSE3

Security

- Encrypted storage of repository passphrases
- Secure session management with cookies
- CSRF protection on all forms
- Input validation and sanitization
- SQL injection prevention via ORM

Performance

- Async/await throughout the application
- Background job processing
- Streaming file downloads
- Efficient FUSE-based archive mounting
- Connection pooling for database access

Known Issues

- Archive browsing requires FUSE support (disabled without it)
- Single-user authentication (multi-user planned)
- Session-only API authentication (API keys planned)

Migration Notes

This is the initial stable release. Future versions will include migration guides for any breaking changes.

Development

- Python 3.11+ requirement
- Modern packaging with pyproject.toml
- Comprehensive test suite with pytest

- Code quality tools (Ruff, MyPy)
- GitHub Actions CI/CD pipeline
- Docker multi-stage builds

Documentation

- Complete user documentation
- API reference with examples
- Installation and deployment guides
- Troubleshooting and how-to guides
- Architecture and contributing documentation

For detailed information about features and usage, see the full documentation.

For the latest changes and development updates, visit the GitHub repository.

5.10 Contributing to Borgitory

Thank you for your interest in contributing to Borgitory! This guide will help you get started with contributing to the project.

5.10.1 Getting Started

Code of Conduct

By participating in this project, you agree to abide by our Code of Conduct. Please be respectful and constructive in all interactions.

Ways to Contribute

- **Bug Reports** - Help us identify and fix issues
- **Feature Requests** - Suggest new functionality
- **Documentation** - Improve guides, API docs, and examples
- **Code Contributions** - Fix bugs, implement features, improve performance
- **Testing** - Write tests, test new features, report compatibility issues
- **Community Support** - Help other users in discussions and issues

5.10.2 Development Setup

Prerequisites

- Python 3.11 or higher
- Git
- BorgBackup installed and in PATH
- Rclone (optional, for cloud sync features)
- Docker (optional, for containerized development)

Setting Up the Development Environment

1. **Fork and Clone the Repository**

```
# Fork the repository on GitHub, then clone your fork
git clone https://github.com/YOUR_USERNAME/Borgitory.git
cd Borgitory
```

2. **Create a Virtual Environment**

```
# Create virtual environment
python -m venv .env_borg

# Activate virtual environment
# On Windows:
.env_borg\Scripts\activate
# On macOS/Linux:
source .env_borg/bin/activate
```

3. **Install Dependencies**

```
# Install in development mode with all dependencies
pip install -e .[dev]
```

4. **Set Up Pre-commit Hooks** (Optional but Recommended)

```
# Install pre-commit
pip install pre-commit

# Install hooks
pre-commit install
```

5. **Run the Development Server**

```
# Start development server with auto-reload
python run.py

# Or using the CLI
borgitory serve --reload --log-level debug
```

6. **Verify Installation**

   - Open http://localhost:8000 in your browser
   - Create a test account and repository
   - Run the test suite: `pytest`

5.10.3 Development Workflow

Branch Strategy

- **main** - Stable release branch
- **develop** - Integration branch for new features
- **feature/*** - Feature development branches
- **bugfix/*** - Bug fix branches

- **hotfix/*** - Critical fixes for production

Workflow:

1. Create a feature branch from `develop`
2. Make your changes
3. Write or update tests
4. Ensure all tests pass
5. Submit a pull request to `develop`

Making Changes

Before You Start:

1. Check existing issues and pull requests
2. Create an issue for discussion (for significant changes)
3. Ensure you understand the project architecture

Development Guidelines:

1. **Follow Code Style**

```
# Run linting
ruff check .

# Auto-fix issues
ruff check . --fix

# Type checking
mypy src/
```

2. **Write Tests**

```
# Run all tests
pytest

# Run with coverage
pytest --cov=borgitory

# Run specific test file
pytest tests/test_repositories.py
```

3. **Update Documentation**
   - Update docstrings for new functions/classes
   - Update README if needed
   - Add entries to CHANGELOG.md
   - Update API documentation if applicable

5.10.4 Code Style Guidelines

Python Code Style

We follow PEP 8 with some modifications:

- **Line length**: 88 characters (Black default)
- **Imports**: Use isort for import sorting
- **Type hints**: Required for all public functions
- **Docstrings**: Google-style docstrings

Example:

```python
from typing import Optional, List
from pydantic import BaseModel


class Repository(BaseModel):
    """Repository configuration model.

    Args:
        name: Human-readable repository name
        path: Filesystem path to repository
        passphrase: Encryption passphrase

    Attributes:
        id: Unique repository identifier
        created_at: Creation timestamp
    """

    name: str
    path: str
    passphrase: str
    id: Optional[int] = None

    def validate_path(self) -> bool:
        """Validate repository path exists and is accessible.

        Returns:
            True if path is valid and accessible

        Raises:
            ValueError: If path is invalid
        """
        # Implementation here
        pass
```

Frontend Guidelines

HTML Templates:

- Use semantic HTML elements
- Include ARIA attributes for accessibility
- Follow BEM naming convention for CSS classes

- Use Tailwind utility classes consistently

JavaScript/Alpine.js:

- Keep JavaScript minimal and focused
- Use Alpine.js directives for reactivity
- Prefer HTMX for server interactions
- Comment complex logic

CSS/Tailwind:

- Use Tailwind utility classes
- Create custom components for repeated patterns
- Ensure responsive design
- Test in multiple browsers

5.10.5 Testing Guidelines

Test Structure

```
tests/
├── conftest.py                 # Pytest configuration and fixtures
├── unit/                       # Unit tests
│   ├── test_models.py
│   ├── test_services.py
│   └── test_utils.py
├── integration/                # Integration tests
│   ├── test_api_endpoints.py
│   ├── test_backup_workflow.py
│   └── test_cloud_sync.py
└── fixtures/                   # Test data and fixtures
    ├── repositories.json
    └── sample_configs.py
```

Writing Tests

Unit Tests:

```python
import pytest
from unittest.mock import Mock, patch
from borgitory.services.borg_service import BorgService


class TestBorgService:
    """Test BorgService functionality."""

    @pytest.fixture
    def borg_service(self):
        """Create BorgService instance for testing."""
        return BorgService()

    def test_create_repository_success(self, borg_service):
```

(continues on next page)

(continued from previous page)

```
        """Test successful repository creation."""
        with patch('subprocess.run') as mock_run:
            mock_run.return_value.returncode = 0

            result = borg_service.create_repository(
                path="/tmp/test-repo",
                passphrase="test-passphrase"
            )

            assert result.success is True
            mock_run.assert_called_once()
```

Integration Tests:

```
import pytest
from httpx import AsyncClient
from borgitory.main import app


@pytest.mark.asyncio
class TestRepositoryAPI:
    """Test repository API endpoints."""

    async def test_create_repository(self):
        """Test repository creation via API."""
        async with AsyncClient(app=app, base_url="http://test") as client:
            response = await client.post(
                "/api/repositories",
                json={
                    "name": "Test Repo",
                    "path": "/tmp/test-repo",
                    "passphrase": "secure-passphrase"
                }
            )

            assert response.status_code == 201
            data = response.json()
            assert data["name"] == "Test Repo"
```

Test Best Practices:

- Write tests for both success and failure cases
- Use descriptive test names
- Keep tests isolated and independent
- Mock external dependencies
- Use fixtures for common test data
- Aim for good test coverage (80%+)

5.10.6 Documentation Guidelines

Documentation Types

- **Code Documentation** - Docstrings and inline comments
- **API Documentation** - OpenAPI/Swagger specs
- **User Documentation** - Installation, usage, and how-to guides
- **Developer Documentation** - Architecture and contributing guides

Writing Documentation

Docstrings:

```
def create_backup(
    self,
    repository: Repository,
    source_path: str,
    archive_name: str,
    compression: str = "lz4"
) -> Job:
    """Create a new backup job.

    Creates a backup job that will archive the specified source path
    to the given repository using BorgBackup.

    Args:
        repository: Target repository for the backup
        source_path: Path to directory or file to backup
        archive_name: Name for the created archive
        compression: Compression algorithm (lz4, zlib, lzma, zstd)

    Returns:
        Job instance representing the backup operation

    Raises:
        ValueError: If source_path doesn't exist
        RepositoryError: If repository is inaccessible

    Example:
        >>> service = BackupService()
        >>> repo = Repository(name="docs", path="/repos/docs")
        >>> job = service.create_backup(repo, "/home/user/docs", "docs-20231201")
        >>> print(f"Backup job created: {job.id}")
    """
```

README Updates:

- Keep installation instructions current
- Update feature lists when adding functionality
- Include relevant badges and links
- Provide clear usage examples

Changelog:

Follow Keep a Changelog format:

```
## [Unreleased]
### Added
- New cloud provider support for DigitalOcean Spaces
- Archive browser file search functionality

### Changed
- Improved backup progress reporting
- Updated Docker base image to Python 3.11

### Fixed
- Fixed repository connection timeout issues
- Resolved FUSE mounting permissions on some systems
```

5.10.7 Submitting Changes

Pull Request Process

1. **Prepare Your Changes**

```
# Ensure your branch is up to date
git checkout develop
git pull upstream develop
git checkout your-feature-branch
git rebase develop
```

2. **Run Quality Checks**

```
# Run all quality checks
python lint.py all

# Run tests
pytest

# Check type hints
mypy src/
```

3. **Commit Your Changes**

```
# Stage changes
git add .

# Commit with descriptive message
git commit -m "feat: add support for DigitalOcean Spaces cloud provider

- Implement DigitalOcean Spaces storage class
- Add configuration validation and tests
- Update provider registry and templates
- Add documentation and examples

Closes #123"
```

4. **Push and Create Pull Request**

```
# Push to your fork
git push origin your-feature-branch

# Create pull request on GitHub
```

Pull Request Guidelines:

- Use a descriptive title
- Include a detailed description of changes
- Reference related issues
- Include screenshots for UI changes
- Ensure all checks pass
- Be responsive to review feedback

Commit Message Format

We follow Conventional Commits:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Types:

- `feat` - New feature
- `fix` - Bug fix
- `docs` - Documentation changes
- `style` - Code style changes (formatting, etc.)
- `refactor` - Code refactoring
- `test` - Adding or updating tests
- `chore` - Maintenance tasks

Examples:

```
feat(cloud-sync): add Azure Blob Storage support

fix: resolve FUSE mounting permission issues on Ubuntu

docs: update installation guide with Docker Compose examples

test: add integration tests for backup scheduling
```

Review Process

What We Look For:

- Code quality and style compliance
- Test coverage for new functionality

- Documentation updates
- Backward compatibility
- Security considerations
- Performance impact

Review Stages:

1. **Automated Checks** - CI/CD pipeline runs
2. **Code Review** - Maintainers review code
3. **Testing** - Manual testing if needed
4. **Approval** - At least one maintainer approval required
5. **Merge** - Squash and merge to develop branch

5.10.8 Community Guidelines

Getting Help

- **GitHub Discussions** - General questions and community support
- **GitHub Issues** - Bug reports and feature requests
- **Documentation** - Check existing docs first
- **Code Review** - Ask questions during review process

Communication

- Be respectful and constructive
- Provide context and examples
- Be patient with responses
- Help others when you can
- Follow up on your issues and PRs

Recognition

Contributors are recognized in:

- **CONTRIBUTORS.md** file
- **Release notes** for significant contributions
- **GitHub contributors** page
- **Special mentions** in project communications

5.10.9 Project Maintenance

Release Process

1. **Version Planning** - Decide on next version features
2. **Feature Freeze** - Stop adding new features
3. **Testing** - Comprehensive testing of release candidate
4. **Documentation** - Update all documentation

5. **Release** - Tag version and publish
6. **Post-Release** - Monitor for issues and feedback

Release Schedule:

- **Major releases** - Every 6-12 months
- **Minor releases** - Every 2-3 months
- **Patch releases** - As needed for critical fixes

Issue Triage

Issue Labels:

- `bug` - Something isn't working
- `enhancement` - New feature or improvement
- `documentation` - Documentation related
- `good first issue` - Good for newcomers
- `help wanted` - Community help requested
- `priority: high` - Critical issues
- `status: needs-info` - Waiting for more information

Triage Process:

1. **Initial Review** - Validate issue and add labels
2. **Prioritization** - Assign priority and milestone
3. **Assignment** - Assign to maintainer or contributor
4. **Progress Tracking** - Monitor progress and provide updates
5. **Resolution** - Close when fixed and verified

5.10.10 Thank You

Your contributions make Borgitory better for everyone. Whether you're fixing a typo, adding a feature, or helping other users, every contribution is valuable and appreciated.

For questions about contributing, please:

- Check the existing documentation
- Search existing issues and discussions
- Create a new discussion or issue
- Reach out to maintainers

Happy contributing!

5.11 License

Borgitory is released under the MIT License.

5.11.1 MIT License

Copyright (c) 2024 mlapaglia

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

5.11.2 Third-Party Licenses

Borgitory uses several third-party libraries and tools. Below are their licenses:

Python Dependencies

- **FastAPI** - MIT License
- **SQLAlchemy** - MIT License
- **Pydantic** - MIT License
- **APScheduler** - MIT License
- **Jinja2** - BSD 3-Clause License
- **Passlib** - BSD License
- **Cryptography** - Apache License 2.0 / BSD License
- **Uvicorn** - BSD 3-Clause License
- **HTTPX** - BSD 3-Clause License
- **Alembic** - MIT License
- **Aiofiles** - Apache License 2.0

External Tools

- **BorgBackup** - BSD 3-Clause License
- **Rclone** - MIT License
- **FUSE** - GPL v2 / LGPL

Frontend Libraries

- **HTMX** - BSD 2-Clause License
- **Alpine.js** - MIT License
- **Tailwind CSS** - MIT License

Development Tools

- **Pytest** - MIT License
- **Ruff** - MIT License
- **MyPy** - MIT License
- **Docker** - Apache License 2.0

5.11.3 License Compatibility

The MIT License is a permissive license that is compatible with most other open source licenses. This means:

- **Commercial Use** - You can use Borgitory in commercial projects
- **Modification** - You can modify the source code
- **Distribution** - You can distribute copies of the software
- **Private Use** - You can use the software privately
- **Patent Use** - Patent rights are granted for the software

Requirements: * Include the original license and copyright notice * Include the license notice in all copies or substantial portions

Limitations: * No warranty is provided * Authors are not liable for damages * No trademark rights are granted

5.11.4 Attribution

When using Borgitory in your projects, we appreciate (but don't require) attribution. You can:

- Link to the project repository: https://github.com/mlapaglia/Borgitory
- Mention Borgitory in your documentation
- Include the project in your credits or about section

5.11.5 Contributing

By contributing to Borgitory, you agree that your contributions will be licensed under the same MIT License that covers the project. See the *Contributing to Borgitory* guide for more information.

5.11.6 Questions

If you have questions about the license or need clarification on usage rights, please:

- Check the MIT License FAQ
- Review the Open Source Initiative resources
- Create an issue on the GitHub repository

The MIT License is widely understood and accepted in the open source community, making Borgitory suitable for use in a wide variety of projects and environments.